Java虚拟机精讲

高翔龙 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

内 容 简 介

HotSpot VM 是目前市面上高性能 JVM 的代表作之一，它采用解释器+JIT 编译器的混合执行引擎，使得 Java 程序的执行性能从此有了质的飞跃。本书以极其精练的语句诠释了 HotSpot VM 的方方面面，比如：字节码的编译原理、字节码的内部组成结构、通过源码的方式剖析 HotSpot VM 的启动过程和初始化过程、Java 虚拟机的运行时内存、垃圾收集算法、垃圾收集器（重点讲解了 Serial 收集器、ParNew 收集器、Parallel 收集器、CMS（Concurrent-Mark-Sweep）收集器和 G1（Garbage-First）收集器）、类加载机制，以及 HotSpot VM 基于栈的架构模型和执行引擎（解释器的工作流程、JIT 编译器的工作流程、分层编译策略、热点探测功能）等技术。

如果你对 JVM 感兴趣，并且从未接触过 JVM，那么本书将会是你探索 JVM 世界的必备入门工具。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Java 虚拟机精讲 / 高翔龙编著. —北京：电子工业出版社，2015.5
ISBN 978-7-121-25705-6

Ⅰ. ①J… Ⅱ. ①高… Ⅲ. ①JAVA 语言—程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2015）第 050785 号

责任编辑：孙学瑛
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：17.5　字数：448 千字
版　　次：2015 年 5 月第 1 版
印　　次：2015 年 5 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前言

大部分 Java 开发人员，除会在项目中使用到与 Java 平台相关的各种高精尖技术，对于 Java 技术的核心 Java 虚拟机了解甚少。这其中最主要的原因或许是在实际的开发过程中，开发人员根本没有机会或是没有必要与 Java 虚拟机等底层技术打交道，更多的只是简简单单地将 Java 虚拟机作为载体，让程序能够顺利运行其上即可。

笔者在面试的过程中，经常会对面试者询问一些与 Java 虚拟机相关的技术问题，但大部分开发人员对于笔者所提及的问题，几乎都有一个统一的答案，那就是不知道、不清楚。有些刚从校门出来的应届毕业生，甚至还包括一些有一定工作经验的开发人员，打心眼儿里觉得 Struts、Spring 和 Hibernate 等上层技术才是重点，基础技术并不重要，这其实是一种本末倒置的“病态”。这就好比金庸武侠小说《笑傲江湖》里的令狐冲，尽管独孤九剑看上去很酷炫，但由于本身内功修炼得不够，与敌人对抗时又能撑得了几个回合呢？

对于那些成熟的第三方开源产品，笔者始终只是把它们当作一种工具，用的时候是宝，不用的时候就丢弃，尽管有些残忍，但确实就是这么现实。以互联网项目为例，对于高性能和稳定性的要求往往大于企业级项目的规范化和流程化，因此架构师每天都会不停地思考，应该如何让我的系统更快？如何避免系统出现瓶颈？如果换作你来做架构师并负责解决这些问题，假如你对 Java 虚拟机一无所知，那么可想而知，你又有什么能力做到系统的性能调优？毕竟性能调优不仅仅只是单纯地从应用代码结构上进行调整，也不是纯粹地依靠物理堆机就能够解决的。

创作此书的目的

尽管并不是所有开发人员都能够在实际的项目开发过程中用到与 Java 虚拟机相关的优化技术，但这并不能够成为你不去了解 Java 虚拟机的理由。只要你从事的是与 Java 开发相关的

岗位，那么对 Java 虚拟机实现机制的了解就是你迟早必须攀爬的一座高山。如果你害怕，那么你将永远也无法屹立在山顶遥望最美的日出。

其实这几年国内也不乏一些比较优秀的技术作者创作了一些知名度比较高的有关 Java 虚拟机的作品。但是这些作品的创作初衷完全不同，一些完全是以理论为重心，而另外一些则完全是从底层源码实战出发为读者诠释 Java 虚拟机的实现细节。尽管这些作品都非常优秀，但笔者认为还不够简单，毕竟所面向的读者更多的是对 Java 虚拟机有一定程度了解的开发人员，对于从未接触过 Java 虚拟机的读者，或许会感觉到有些手足无措的挫败感。因此笔者创作此书的目的，在更大程度上是以更为精练的语句引读者入门 Java 虚拟机的世界。换句话说，你完全可以将本书看作一个跳板，当你熟知本书的内容后，如果有一种饥渴难耐的感觉，那么恭喜你，笔者建议你阅读难度更大的书籍，同时本书的目的也就达到了。

本书所面向的读者

本书适用于任何对 Java 虚拟机感兴趣的 Java 开发人员、系统架构师、Java 虚拟机爱好者。尤其是对于那些从未接触过 Java 虚拟机的 Java 开发人员，本书笔者竭尽所能用最精练和直接的语句诠释了有关 Java 虚拟机的方方面面，只要你熟练掌握了 Java 编程基础，那么阅读本书你将不会感觉到任何的吃力和枯燥乏味。

本书内容

本书的内容包括字节码的编译原理、字节码的内部组成结构、通过源码的方式剖析 HotSpot VM 的启动过程和初始化过程、Java 虚拟机的运行时内存、垃圾收集算法、垃圾收集器［重点讲解了 Serial 收集器、ParNew 收集器、Parallel 收集器、CMS（Concurrent-Mark-Sweep）收集器和 G1（Garbage-First）收集器］、类加载机制，以及 HotSpot VM 基于栈的架构模型和执行引擎（解释器的工作流程、JIT 编译器的工作流程、分层编译策略、热点探测功能）等技术。

从本书的第 1 章开始，笔者首先对 Java 的体系结构做了一个简单且全面的介绍，让大家深刻认识到了 Java 虚拟机在 Java 平台中所占的分量，然而这一章的重点则是在 OpenJDK 和 HotSpot VM 的编译实战任务上。当大家对 Java 虚拟机有了一个简单的了解后，本书的第 2 章则开始对字节码的编译原理进行了讲解，因为大家有必要了解 Java 语言规范与 JVM 规范之间的区别，以及 Java 代码究竟需要经历哪些步骤之后才能够被编译为一个有效的字节码文件。而本书的第 3 章则与上一章息息相关，当大家了解字节码的编译原理后，接下来笔者将会对字节码的内部组成结构进行深入讲解。本书的第 4～6 章涉及 HotSpot VM 的部分源码实现，从 Launcher 启动 HotSpot VM 开始，到 HotSpot VM 的初始化过程都是每一个 Java 开发人员必须

掌握和了解的。对于大部分 Java 开发人员而言，对 Java 虚拟机最感兴趣的内容莫过于内存管理和垃圾收集，本书用了大量的篇幅来对这些技术进行讲解，并且对 Java7 新增的 G1 收集器也做了介绍。在本书的第 7 章中，笔者对类加载机制进行了讲解，帮助大家对类的初始化过程理解透彻。然而在本书的最后一章中，笔者对 HotSpot VM 的架构模型和执行引擎进行了深入讲解，毕竟执行引擎是 Java 虚拟机中最重要同时也是最核心的部分，运行时编译技术使得 Java 程序的运行性能从此有了质的飞越。

参考文献

笔者在本书的创作过程中，从下面所列的这些参考资料中获取了极大的帮助，大家同时也可以通过如下信息找到更多关于 Java 虚拟机方面的资料，毕竟单靠一本书就想了解 Java 虚拟机的所有技术细节几乎是不可能的，更是不现实的。

- 《Java 虚拟机规范 Java SE7 版》 [美] Tim Lindholm、Frank Yellin、Gilad Bracha、Alex Buckley 著；
- 《深入 Java 虚拟机 第 2 版》 [美] Bill Venners 著；
- 《Java 性能优化权威指南》 [美]Charlie Hunt、Binu John 著；
- 《深入理解 Java 虚拟机 第 2 版》周志明 著；
- 《HotSpot 实战》 陈涛 著；
- 《程序员》2014 年 3 月刊《中间语言和虚拟机漫谈》 徐宥 著；
- 《JVM 分享：Java Program in Action》 Rednaxelafx（莫枢） 著；
- 《HotSpot 内存管理白皮书》。

感谢

此书献给我这辈子最爱的姥爷。从我诞生那天起，您就将您的慈爱毫不吝啬地给予了我，感谢您和姥姥从小对我的陪伴和照顾，是你们让我拥有了愉快的童年，让我体会到了亲情的温暖。我记得小时候家里的鹦鹉、画眉鸟是您最爱的宠物，您会每天带着它们出去散步，而如今却成为了我心中永远的回忆。姥爷，2014 年 05 月 09 日早上 7 点 20 分，当我最后在您耳边说完悄悄话后，您走了，带走了我的思念！姥爷，我一定会成为您这辈子的骄傲！除此之外，我们家可爱的小娇娇同学，谢谢你的支持和鼓励才让我有创作此书的勇气和动力，谢谢你，我爱你！

接下来要感谢的是 Rednaxelafx（莫枢）和蔡学镛，您二位在百忙之中抽空阅读了本书，提出了许多宝贵的意见，没有你们的支持或许本书至今也无法顺利出版。

最后还要感谢的是电子工业出版社博文视点的所有编辑们，谢谢你们无条件忍受着我一再的跳票，本书能够顺利出版离不开你们如此敬业的精神和一丝不苟的工作态度，由衷地谢谢你们。

高翔龙

2015 年 3 月

目录

第 1 章　Java 体系结构 …… 1

1.1　认识 Java …… 1

1.1.1　与生俱来的优点 …… 2

1.1.2　语法结构和对象模型 …… 4

1.1.3　历史版本追溯 …… 5

1.2　Java 重要概念 …… 7

1.2.1　Java 编程语言 …… 7

1.2.2　字节码 …… 7

1.2.3　Java API …… 8

1.2.4　Java 虚拟机 …… 8

1.3　安装与配置 Java 运行环境 …… 10

1.3.1　Windows 环境下的安装与配置 …… 10

1.3.2　Linux 环境下的安装与配置 …… 11

1.3.3　编写 Java 程序 …… 12

1.3.4　编译与运行 …… 13

1.3.5　关键字与标示符 …… 13

1.4　Java 技术的新特性 …… 14

1.4.1　Java 模块化与 OSGi 技术 …… 14

1.4.2　语言无关性 …… 15

1.4.3　使用 Fork/Join 框架实现多核并行 …… 16

1.4.4　丰富的语法特性 …… 17

1.4.5　过渡到 64 位虚拟机 …… 18

1.5　实战：玩转 OpenJDK …… 19

1.5.1　JDK 与 OpenJDK 的关系 …… 19

1.5.2　基于 OpenJDK 深度定制的淘宝 JVM（TaobaoVM） …… 20

1.5.3　下载 OpenJDK 源代码 …… 22

1.5.4　构建编译环境 …… 22

1.5.5　执行整个 OpenJDK 的编译 …… 23

1.5.6　执行单独 HotSpot 的编译 …… 26

1.5.7　导致编译失败的一些疑难杂症 …… 29

1.5.8　使用 GDB 工具 Debug HotSpot …… 30

1.6　本章小结 …… 36

第 2 章　字节码的编译原理 …… 37

2.1　javac 编译器简介 …… 37

2.1.1　javac 与 Eclipse Compiler for Java 编译器 …… 38

2.1.2　javac 的使用与标准选项配置 …… 39

2.1.3　编译原理 …… 40

2.1.4　下载 javac 编译器源码 …… 41

2.1.5　调用 compile()方法执行编译 …… 41

2.2　词法解析步骤 …… 43

2.2.1 Token 序列 …… 45
2.2.2 源码字符集合与 Token 之间的对应关系 …… 47
2.2.3 调用 key()方法获取指定 Token …… 48
2.2.4 调用 nextToken()方法计算 Token 的获取规则 …… 48
2.2.5 调用 parseCompilationUnit()方法执行词法解析 …… 49
2.3 语法解析步骤 …… 51
2.3.1 调用 qualident()方法解析 package 语法节点 …… 52
2.3.2 调用 importDeclaration()方法解析 import 语法树 …… 54
2.3.3 调用 classDeclaration()方法解析 class 语法树 …… 56
2.4 语义解析步骤 …… 59
2.5 生成字节码 …… 61
2.6 实战：使用 javap 工具分析字节码 …… 62
2.7 实战：使用 GCJ 编译器将 Java 源码直接编译为本地机器指令 …… 64
2.8 本章小结 …… 66
第 3 章 字节码文件 …… 67
3.1 字节码文件的内部组成结构 …… 67
3.2 符号引用 …… 73
3.2.1 类或者接口的全限定名 …… 74
3.2.2 简单名称 …… 74
3.2.3 描述符 …… 74
3.3 常量池 …… 76
3.3.1 CONSTANT_Utf8_info 常量项 …… 77
3.3.2 CONSTANT_Integer_info 常量项 …… 77
3.3.3 CONSTANT_Float_info 常量项 …… 78
3.3.4 CONSTANT_Long_info 常量项 …… 78
3.3.5 CONSTANT_Double_info 常量项 …… 79
3.3.6 CONSTANT_Class_info 常量项 …… 79
3.3.7 CONSTANT_String_info 常量项 …… 80
3.3.8 CONSTANT_Fieldref_info 常量项 …… 81
3.3.9 CONSTANT_Methodref_info 常量项 …… 81
3.3.10 CONSTANT_InterfaceMethodref_info 常量项 …… 82
3.3.11 CONSTANT_NameAndType_info 常量项 …… 82
3.3.12 CONSTANT_MethodHandle_info 常量项 …… 83
3.3.13 CONSTANT_MethodType_info 常量项 …… 84
3.3.14 CONSTANT_InvokeDynamic_info 常量项 …… 84
3.4 字段表 …… 85
3.5 方法表 …… 86
3.6 属性表 …… 88
3.6.1 Code 属性 …… 89
3.6.2 ConstantValue 属性 …… 90
3.6.3 Exceptions 属性 …… 91
3.6.4 LineNumberTable 属性 …… 92
3.6.5 SourceFile 属性 …… 93
3.6.6 LocalVariableTable 属性 …… 93
3.6.7 InnerClasses 属性 …… 94
3.6.8 BootstrapMethods 属性 …… 95
3.7 本章小结 …… 96

第 4 章 剖析 HotSpot 的 Launcher ····· 97
4.1 HotSpot 的源码目录结构 ····· 97
4.2 Launcher 简介 ····· 99
4.3 跟踪 Launcher 的执行过程 ····· 101
4.3.1 使用 Launcher 启动 JVM ····· 101
4.3.2 启动函数 main() ····· 102
4.3.3 在主线程中执行 JavaMain() 函数 ····· 106
4.3.4 调用 JNI_CreateJavaVM() 函数初始化 HotSpot ····· 114
4.3.5 调用 LoadClass()函数获取 Java 启动类 ····· 115
4.3.6 调用 GetStaticMethodId() 函数获取 Java 启动方法 ····· 116
4.3.7 调用 CallStaticVoidMethod() 函数执行 Java 启动方法 ····· 116
4.3.8 调用 jni_DestroyJavaVM 函数销毁 HotSpot ····· 119
4.4 实战：在 Launcher 中添加自定义函数模块 ····· 120
4.5 本章小结 ····· 121
第 5 章 剖析 HotSpot 的初始化过程 ····· 122
5.1 HotSpot 的构成模块 ····· 122
5.2 Prims 模块 ····· 124
5.2.1 JNI 子模块 ····· 124
5.2.2 JVM 子模块 ····· 125
5.2.3 JVMTI 子模块 ····· 128
5.2.4 Perf 子模块 ····· 129
5.3 Runtime 模块 ····· 129
5.3.1 Thread 子模块 ····· 131
5.3.2 调用 create_vm()函数完成 HotSpot 的最终初始化 ····· 131
5.4 跟踪 HotSpot 的初始化过程 ····· 140
5.4.1 调用 init()和 init_2()函数初始化 os 模块 ····· 141
5.4.2 调用 vm_init_globals()函数初始化全局数据结构 ····· 144
5.4.3 调用 init_globals()函数初始化全局模块 ····· 144
5.5 本章小结 ····· 146
第 6 章 内存分配与垃圾回收 ····· 147
6.1 JVM 的运行时内存区结构 ····· 147
6.2 线程共享内存区 ····· 148
6.2.1 Java 堆区 ····· 148
6.2.2 方法区 ····· 150
6.2.3 运行时常量池 ····· 150
6.3 线程私有内存区 ····· 150
6.3.1 PC 寄存器 ····· 151
6.3.2 Java 栈 ····· 151
6.3.3 本地方法栈 ····· 152
6.4 性能监控区 ····· 152
6.5 自动内存管理 ····· 152
6.5.1 内存分配原理 ····· 153
6.5.2 逃逸分析与栈上分配 ····· 157
6.5.3 对象内存布局与 OOP-Klass 模型 ····· 158
6.5.4 GC 的作用 ····· 159
6.5.5 垃圾标记：根搜索算法 ····· 160
6.5.6 垃圾回收：分代收集算法 ····· 161
6.6 垃圾收集器 ····· 164
6.6.1 串行回收：Serial 收集器 ····· 165
6.6.2 并行回收：ParNew 收集器 ··· 166
6.6.3 程序吞吐量优先：Parallel 收集器 ····· 166
6.6.4 低延迟：CMS（Concurrent-Mark-Sweep）收集器 ····· 167
6.6.5 区域化分代式：G1（Garbage-First）收集器 ····· 170

6.6.6 垃圾收集的相关选项配置……172
6.7 实战：GC 日志分析……175
6.7.1 不同 GC 日志的展示形式……175
6.7.2 使用 GCHisto 工具分析离线日志……179
6.8 实战：分析 dump 文件……181
6.8.1 使用 jmap 工具生成 dump 文件……181
6.8.2 使用 MAT（Memory Analyzer Tool）工具分析 dump 文件…182
6.9 本章小结……184
第 7 章 类加载机制……185
7.1 类加载器……185
7.1.1 抽象类 ClassLoader……187
7.1.2 双亲委派模型……188
7.1.3 自定义类加载器……191
7.1.4 定位 ClassNotFoundException 异常……193
7.1.5 定位 NoClassDefFoundError 异常……194
7.2 类的加载过程……195
7.2.1 加载字节码……198
7.2.2 验证阶段……199
7.2.3 准备阶段……200
7.2.4 解析阶段……201
7.2.5 初始化阶段……201
7.3 实战：字节码文件的加密与解密……204
7.4 本章小结……208
第 8 章 剖析 HotSpot 的架构模型与执行引擎……209
8.1 栈帧的组成结构……209
8.1.1 局部变量表……211
8.1.2 操作数栈……212
8.1.3 动态链接……214
8.1.4 方法返回值……216
8.2 HotSpot 中执行引擎的架构模型……216
8.2.1 本地机器指令……217
8.2.2 寄存器架构与栈式架构之间的区别……218
8.2.3 基于栈式架构的设计……221
8.2.4 调用 call_stub()函数执行 Java 方法……222
8.2.5 栈顶缓存（Top-of-Stack Cashing）技术……225
8.2.6 实战：跟踪字节码解释器的执行步骤……227
8.3 解释器与 JIT 编译器……230
8.3.1 查阅 HotSpot 的运行时执行模式……231
8.3.2 解释器的工作机制与构成模块……232
8.3.3 JIT 编译器的工作机制与构成模块……234
8.3.4 分层编译策略……235
8.3.5 热点探测功能……236
8.4 本章小结……239
附录 A Java7 新增语法特性……241
A.1 try-with-resources 语句……241
A.2 泛型的“<>”类型推断运算符…245
A.3 声明二进制字面值……247
A.4 字面值下画线支持……248
A.5 switch 表达式支持 String 类型…250
A.6 mutil-catch 特性……251
A.7 NIO2.0 文件系统的改变……255
附录 B 指令助记符……262

第 1 章

Java体系结构

1.1 认识Java

经历了多年的发展，Java早已由一门单纯的计算机编程语言，演变为一套强大的技术体系平台。根据不同的技术规范，Java设计者们将Java划分为 3 种结构独立但却又彼此依赖的技术体系分支，分别是Java SE（标准版）、Java EE（企业版）和Java ME（精简版）。在此大家需要注意，本书所提及的这 3 种技术体系分支，分别对应着不同的规范集合和组件。Java SE活跃在桌面领域，主要包含了Java API组件。而Java EE则活跃在企业级领域，除了包含Java API组件外，还扩充有Web组件、事务组件、分布式组件、EJB组件、消息组件等；综合这些技术，开发人员完全可以构建出一个具备高性能、结构严谨的企业级应用，并且Java EE也是用于构建SOA[①]架构的首选平台。至于Java ME则活跃在嵌入式领域，之所以将其称之为精简版，那是因为该平台仅保留了Java API中的部分组件，以及适应设备的一些特有组件。

Java在奠定了企业级领域的霸主地位后，目前正一步步朝着移动领域的方向大展拳脚，这不仅要感谢移动互联网的迅速崛起，还得多亏Google选择Java作为Android操作系统的应用层编程语言。就目前而言，Java已经成为了全球开发人员使用最为广泛的一种编程语言。从随处可见的手持移动设备、嵌入式设备、个人电脑、高性能的集群服务器或大型机中，我们几乎随处都可以看见Java程序的身影。或许当你还在犹豫和怀疑Java能做什么的时候，Java早已在企业级领域、互联网领域、移动领域、中间件领域，甚至是游戏领域都发展得如火如荼。比如著名的开源 3D游戏引擎jME（j-Monkey-Engine）[②]就是完全采用Java语言编写的，

① SOA（Service-Oriented-Architecture，面向服务架构）作用于分布式的系统集成环境中，它将程序的内部功能通过定义良好的契约对外发布成体系结构中立的接口，以此满足不同系统之间的交互操作。

② jME（j-Monkey-Engine）官方地址：http://www.jmonkeyengine.org/。

该引擎可以算是目前Java平台上最流行，同样也是应用最广泛的 3D游戏引擎。当然这所有的一切都离不开Java的运行支撑系统，那就是Java虚拟机，Java与生俱来的通用性、安全性和高效性都建立在Java虚拟机之上。

从早期版本到每一个新版本的迭代，Java 都会不断完善自身缺陷，并进行语法增强，这无疑是带给开发人员最好的礼物。本书不仅会重点讲解与 Java 虚拟机相关的一些知识点，在本书的附录中笔者还为大家讲解了有关 Java7 在语法层面上的一些改变和扩充，让大家更全面地了解和掌握 Java 技术。

1.1.1 与生俱来的优点

面向对象的思想如今已经渗透到软件开发的各个领域，例如 OOA（Object Oriented Analysis，面向对象的分析）、OOD（Object Oriented Design，面向对象的设计），以及开发人员时常挂在嘴边的 OOP（Object Oriented Programming，面向对象的编程）。除了在急需注重性能与效率的应用场景下，开发人员大多数时候都是在使用面向对象等高级语言，比如 C#、C++、Ruby、PHP 等。这些高级语言无论是从设计原理或者是从实现细节上来看都是非常精妙的，那么与这些同样优秀的语言相比，Java 的优势主要体现在哪里呢？本书归纳了 Java 的 5 项重要优势：

- □ 体系结构中立；
- □ 安全性优越；
- □ 多线程；
- □ 分布式；
- □ 丰富的第三方开源组件。

Java 之所以能够实现“一次编译，处处运行”（Write Once，Run Anywhere），功不可没的首先当属字节码。和 C/C++等传统的编译性语言不同，Java 源代码的默认编译结果并非是可执行代码（本地机器指令），而是具有平台通用性的字节码。尽管不同平台 Java 虚拟机的内部实现机制不尽相同，但是它们共同解释出的字节码却是一样的，所以说字节码才是 Java 实现跨平台的关键要素，如图 1-1 所示。体系结构中立不仅使得 Java 天生具备跨平台的优势，同时还延伸了程序的安全性，因为 Java 程序始终只能够运行在 Java 虚拟机中，这与实际的物理宿主环境之间是相互“隔离”的，换句话说 Java 的安全模型可以禁止很多不安全的因素，有助于防止错误的发生，增强程序的可靠性。当然 Java 的部分语法限制也在某种意义上保障了程序的安全，比如废弃指针操作、自动内存管理、数组边界检查、类型转换检查、线程安全机制和物理环境访问限制等。

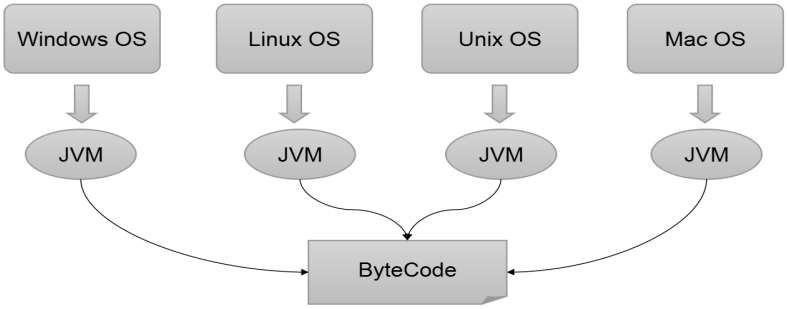


图 1-1 Java 跨平台特性

开发人员往往希望由自己编写的程序能够在性能上满足高效执行的要求，那么这就需要在程序中正确运用多线程技术去有效地并发执行操作任务。单线程是按照顺序结构进行串行执行的，我们暂且先不讨论程序的执行效率，一旦程序因执行某一个任务而发生异常，往往有可能导致程序终止而无法继续往下执行。当遇到这种情况的时候，就需要一种机制能够将这种任务从主线程中剥离出来，哪怕是发生异常也不会影响系统的整体运行。从另一个角度来说，我们可以使用多线程的并发机制将任务进行分散，而不是全部集中在主线程内，采用异步的方式去并发执行多项任务，这样的程序设计架构必然会有极高的执行效率。

由于单台服务器的处理能力很有限，甚至在某些情况下单台服务器的处理能力还存在性能瓶颈，因此在生产环境中架构师往往会考虑采用分布式架构的方式来部署应用。只有充分利用分布式环境中的每一个节点去协同处理任务，才能够换来较高的执行效率，并且还能有效降低单机负载以及提升稳定性和可用性。Java 与生俱来对分布式技术的支持就比较完善，比如 Java EE 规范中的 RMI（Java Remote Method Invocation，Java 远程方法调用）、JMS（Java Message Service，Java 消息服务）等技术。

框架这个名词相信对于绝大多数开发人员而言并不会感觉到陌生，它是对编码规范的一种抽象。在企业内部开发人员往往会为了提高生产效率和易于扩展从而选择使用框架，因为遵守框架定义的内部契约，不仅可以更高效地解决技术难题，同时还能够缩短项目的开发周期，总之如果能够在项目中正确选择和使用框架，那么所带来的好处将会是不言而喻的。

衍生在Java平台上的各类第三方开源框架几乎随时都保持着更新，可以毫不客气地说，在如今的整个领域模型中，随处可见成熟的第三方开源框架已经开始“泛滥”，当然本书仅以贯穿整个领域模型的Spring框架为例进行介绍。开发人员不仅可以选择使用Spring提供的全部功能，甚至还可以根据业务需求选择使用Spring的部分功能子集。如果想要使用MVC[③]，

③ MVC（Model-View-Controller，模型层-视图层-控制层）是一种编程模型，它将一个系统拆分为 3 大类，视图层仅只负责页面显示和数据显示工作，控制层则负责客户端的请求/响应和业务调用工作，

Spring提供支持RESTful风格的Spring MVC，甚至允许与其他第三方MVC框架进行无缝集成；如果想要使用Spring的核心功能，那么Spring最新发布的 4.x版本将会有全新体验；如果想对权限进行控制管理，Spring提供Spring Security；如果不想手动管理持久层事务，Spring提供丰富的ORM集成策略，或者直接选用Spring JDBC（Spring JDBC仅仅只是针对传统的JDBC做了轻量级封装，熟悉JDBC的开发人员都能够迅速掌握并使用）作为持久层框架；如果想降低使用Hadoop进行海量数据处理所带来的复杂度和学习成本，Spring提供Spring Hadoop；甚至如果你是Android的开发人员，Spring照样提供基于Mobile领域的各类规范和实现。

当然笔者并不是在为 Spring 打广告和做宣传，只是想告诉大家，Java 真正强大的地方是因为拥有全世界最多的技术拥护者和开源社区支持，他们无时无刻都保持着最充沛的体力与思维，一步一步地驱动着 Java 技术的走向，其实这才是 Java 最大的优势和财富。

1.1.2 语法结构和对象模型

Java 继承了 C 语言的语法结构，并改编了 C++语言的对象模型，所以注定了 Java 天生就适合书写优美的代码。我们都知道，类是最基本的封装单元，所有的操作都将发生在类中，那么定义一个类，其内部的组成结构又是怎样的呢？简单来说，属性和方法构成了一个简单的类，属性用于定义对象的各种“器官”，而方法则用于定义对象的一系列“行为”。虽然这样的描述很直观，却显得比较粗糙，如果将其细化分类后，大家或许会发现，Java 语法结构的设计是非常精妙的，同样又不失灵活性和简洁性。在类内部，开发人员可以定义许多元素特征，这些元素都统称为类成员，本书归纳了 Java 的一些基本类元素信息：

- □ 关键字；
- □ 标示符；
- □ 操作符（空白分隔符、普通分隔符）；
- □ 注解（@Annotation 类型、描述类型）；
- □ 数据类型（原始数据类型、引用类型）；
- □ 属性（常量、变量）；
- □ 运算符和表达式；
- □ 控制语句（流程控制语句、循环控制语句）；
- □ 异常处理；
- □ 方法体。

对象模型与面向对象（Object Oriented）的特性之间保持着一种密不可分的关系。面向

而模型层则负责实际的业务操作。

对象之所以目前大行其道，其中最关键的因素在于在系统构建复杂化的当下，允许开发人员以面向对象式思维设计出更具复用性、维护性、扩展性和伸缩性的应用程序。然而在语法层面上，开发人员可以在程序中直接使用 new 关键字创建出一个对象，并返回当前对象的一个引用（reference）。在此大家需要注意，Java 中的引用操作绝不等价于 C++中的指针，因为引用类型的变量持有的仅仅只是一个引用而已而非实际值，也就是说开发人员并不能在程序中直接与对象实例打交道，而必须通过引用进行“牵引”，如图 1-2 所示。在程序中即便不存在对象实例，引用也允许独立存在，也就是说可以声明一个引用，不一定非要有一个对象实例与之关联，但务必确保在真正使用一个对象时，它已经完成了初始化操作，也就是执行了<init>()方法。

图 1-2　引用关系

在谈及如何在语法层面上创建一个对象之后，笔者不得不提及的还有构造方法，构造方法出现的目的就是为了初始化对象以及成员变量。在 Java 中对象的初始化和创建其实是同一个操作，虽然从字义上来理解，初始化和创建并不是同一个概念，但它们却恰巧被 Java 设计者们绑定在一起，谁也离不开谁。并且在继承环境下，派生类与超类之间构造方法的加载顺序同样也是按照派生顺序进行加载的，这样做的目的就是为了确保每一个对象在使用前都已经被成功初始化过。

1.1.3　历史版本追溯

Java 发展至今差不多已经有二十多个年头了，从诞生到如今的茁壮，Java 可谓是具备了天时地利人和。1991 年 Sun 公司的 James Gosling（Java 语言的主要创始人）等人为嵌入式设备开发了一种叫作 Oak（一种橡树的名称）的编程语言，其实 Oak 就是如今 Java 语言的前身，只是在当时的特定环境下，Oak 语言并没有引起大多数人的注意。直到 1994 年，随着互联网和 3W（World Wide Web）的迅猛发展，他们使用 Oak 编写了一个叫作 HotJava 的浏览器，这才得到了 Sun 公司的首席执行官 Scott McNealy 的支持，最终 Java 才得以继续研发和发展。

为了促销和法律等原因，1995 年 Oak 语言正式更名为 Java，同年正式在 Sun World 大会上发布了 Java 1.0 版本，并且首次提出了“Write Once，Run Anywhere”的口号。其实 Java 的得名还有段小插曲，有一天 Oak 小组成员正在喝咖啡时，议论给新语言起个什么名字比较好，于是有人提议用 Java（Java 是印度尼西亚盛产咖啡的一个岛屿）作为新语言的名字，这个提议得到了其他小组成员的赞同，就采用 Java 来命名此新语言。很快 Java 就被工业界

所认可，许多大公司，如 IBM Microsoft.DEC 等购买了 Java 的使用权，并被美国杂志 PC Magazine 评为 1995 年十大优秀科技产品，从此开启了 Java 的新篇章。直至 2010 年 IT 巨头 Oracle 出面收购 Sun 公司，Java 则更是达到了巅峰。或许有部分开发人员经常在论坛“忧愁”Java 的未来，但相信大多数开发人员和笔者一样，都认为 Oracle 收购 Sun 是一件好事，毕竟 Java 的未来还有很长的路要走，Oracle 这样的企业必然具备优秀的技术实力可以为 Java 注入更多、更丰富的特性和变化。

Java1.0 正式版本所包含的功能并不算多，除了在 JDK 中配套一个纯解释器实现的 Java 虚拟机外（Sun Classic VM），仅支持 Applet、AWT 等技术。直到 1997 年 Java1.1 在 1.0 的基础之上添加了 JDBC、JAR 格式支持、JavaBeans、RMI 等技术，并在语法层面上开始支持反射和内部类等操作后，Java 的功能才开始逐渐变得丰富。

1998 年 Sun 公司发布了 Java1.2 版本，并在此版本中将 Java 划分为 3 种结构独立却彼此依赖的技术体系分支，分别是 J2SE（标准版）、J2EE（企业版）和 J2ME（精简版）。在这个版本中也添加了 EJB、Java Plug-in、Java IDL、Swing 等技术，并且 Sun 公司首次在 JDK 中内置了 JIT 编译器。

2000 年 Sun 公司发布了 Java1.3 版本，但这个版本并没有引进太多的技术和改变，只是在基础类库上做了一些改进。直到 2002 年 Java1.4 版本的发布才是重头戏，这意味着 Java 已经开始逐渐走向成熟。Java1.4 发布了相当多的特性，如正则表达式、异常链、NIO、日志类、XML 解析器和 XSLT 转换器等。时至今日国内某些大型企业仍然还在沿用 Java1.4 的版本，并且很多优秀的第三方开源产品同样也针对 Java1.4 版本做了向下兼容。

2004 年 Sun 公司发布了 Java1.5 版本，同时也是在 1.5 版本发布后，后续的 Java 版本都改为以 Java5、Java6、Java7 等规则进行命名。在早期版本中，Sun 公司对 Java 各个版本的语法层面的改变并不大，但 1.5 版本针对语法层面的改进却相当多，几乎导致整个 API 都发生了变化。比如自动装箱/拆箱、泛型、枚举、@Annotation、可变长参数、foreach、粗粒度的并行模型等。

2006 年 Sun 公司发布了被 Oracle 收购之前的最后一个版本 Java6。在该版本中，Sun 公司改变了从 Java1.2 开始的惯用的 J2SE（标准版）、J2EE（企业版）和 J2ME（精简版）命名方式，更名为 Java SE（标准版）、Java EE（企业版）和 Java ME（精简版）。Java6 的改变更多是体现在虚拟机内部，主要以同步对象锁、垃圾回收、类型装载等方面的算法更新为主。同年 Sun 公司正式宣布 Java 以 GPL（General Public License）v2 的开源协议进行源代码公开，并建立了 OpenJDK 对 Java 的源代码库进行独立管理。

由于 Sun 公司无力推动 Java7 的研发工作，2010 年正式被 Oracle 公司收购，并由 Oracle 正式接替 Java7 的后续研发工作。由于 Java7 预期的功能非常多，这不得不导致 Oracle 将部

分功能进行裁剪，延迟到 Java8 的版本中再进行发布。Java7 的正式版本不仅在语法层面上做了较大改变，还引入了许多新的技术，比如更新了 Java 的文件系统、细粒度的 Fork/Join 并行编程、混合语言等。同时 Java 虚拟机内部也做了许多改进和调整，比如 Java7 提供了 G1 垃圾收集器、类装载器的并行装载增强实现等。

1.2 Java重要概念

Java 的体系结构主要由 Java 编程语言、字节码、Java API 和 Java 虚拟机等 4 部分独立却相关的技术组成。或许很多时候我们并没有刻意去关注它们，但确确实实当我们在编写 Java 程序的时候，就同时用到了这 4 种技术。首先我们使用 Java 编程语言编写好 Java 程序的源代码，然后 Java 前端编译器负责将 Java 源代码编译为字节码，接着 Java 虚拟机负责将这些编译好的字节码装载进内部，最后解释/编译为对应平台上的机器指令运行。这就是一个完整的 Java 程序从编写到最终执行的结构链路。

1.2.1 Java编程语言

Java 编程语言是一种语法结构严谨、体系结构中立、面向对象、支持多核并行的程序设计语言，它继承了 C 语言的语法结构，并改编了 C++的对象模型，所以 Java 天生就适合书写优美的代码。并且 Java 舍弃了 C 和 C++中许多不安全的语法特性，比如：废弃指针操作、自动内存管理、数组边界检查、类型转换检查、线程安全机制和物理环境访问限制等。

1.2.2 字节码

Java最初诞生的目的就是为了在不依赖于特定的物理硬件和操作系统环境下运行，那么也就是说Java程序实现跨平台特性的基石其实就是字节码。Java之所以能够解决程序的安全性问题、跨平台移植性等问题，最主要的原因就是Java源代码的编译结果并非是本地机器指令 [④]，而是字节码。当Java源代码成功编译成字节码后，如果想在不同的平台上面运行，则无须再次编译，也就是说Java源码只需一次编译就可处处运行，这就是“Write Once，Run Anywhere”的思想。所以注定了Java程序在任何物理硬件和操作系统环境下都能够顺利运行，只要对应的平台装有特定的Java运行环境，Java程序都可以运行，虽然各个平台的Java虚拟机内部实现细节不尽相同，但是它们共同执行的字节码内容却是一样的。

④ 本地机器指令：可以被计算机 CPU 直接进行识别并执行，其表现形式为二进制。机器指令通常由操作码和操作数两部分组成，操作码负责指令需要完成的操作，即指令的功能。而操作数负责数据运算，以及运算结果所存放的位置等。

那么什么是字节码（ByteCode）呢？参考《Java 虚拟机规范 Java SE7 版》的描述来看，任何编程语言的编译结果满足并包含 Java 虚拟机的内部指令集、符号表以及一些其他辅助信息，它就是一个有效的字节码文件，就能够被虚拟机所识别并装载运行。在大部分情况下，字节码更多是存储在本地磁盘文件中，比如后缀名为“.class”的文件。每一个字节码文件都对应着全局唯一的一个类或者接口的定义信息，但这也并非绝对，类和接口并不一定都只能存储在文件里，它还可以通过类装载器直接在运行时生成。

字节码结构组成比较特殊，其内部并不包含任何的分隔符区分段落，所以无论是字节顺序、数量都是有严格规定的，所有 16 位、32 位、64 位长度的数据都将构造成 2 个、4 个和 8 个 8 位字节单位来表示，多字节数据项总是按照 big-endian 顺序（高位字节在地址最低位，低位字节在地址最高位）来进行存储。也就是说，一组 8 位字节单位的字节流组成了一个完整的字节码文件。

1.2.3 Java API

API（Application Programming Interface，应用程序编程接口）是一些预先定义的接口，目的是提供应用程序与开发人员基于某软件或硬件的以访问一组例程的能力，而又无需访问源码或理解内部工作机制的细节。Java API 通过支持与平台无关性和安全性，使得 Java 程序适应任何应用场景。那么 Java API 中其实包含的就是 Java 的基础类库集合，它提供一套访问主机系统资源的标准方法。

1.2.4 Java虚拟机

Java 技术的核心就是 Java 虚拟机（JVM，Java Virtual Machine），因为所有的 Java 程序都运行在 Java 虚拟机内部。JVM 之所以被称之为 VM，是因为它是由一组规范所定义出的抽象计算机。JVM 的主要任务就是负责将字节码装载到其内部，解释/编译为对应平台上的机器指令执行，如图 1-3 所示。

Sun 公司的 HotSpot VM 应该是大多数开发人员最熟悉的一款高性能 Java 虚拟机，它是 JDK 和 OpenJDK 中缺省自带的一款虚拟机，同样也是目前市面上应用最广的一款 Java 虚拟机。但这款虚拟机最早是由一家名不见经传的小公司“Longview Technologies”研发设计出来的，后来这家公司被 Sun 公司收购后，HotSpot 虚拟机也同样被纳入麾下。

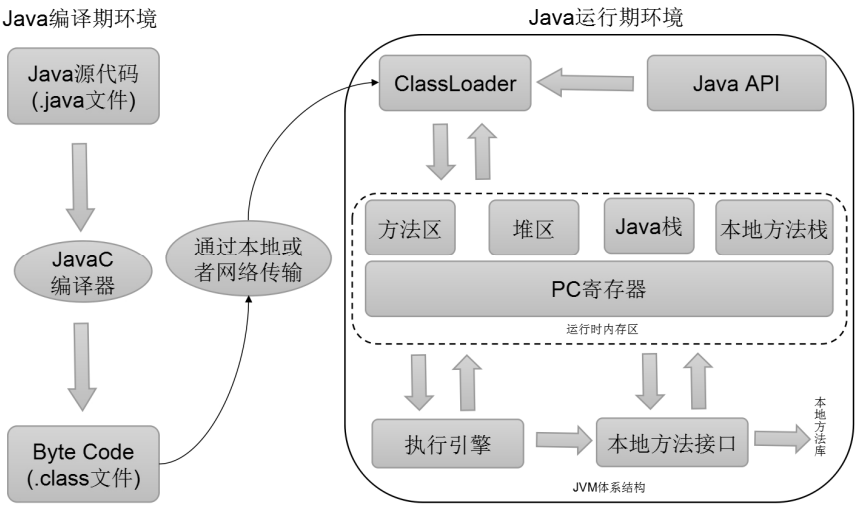


图 1-3 字节码编译期与运行期

HotSpot VM 是目前市面上高性能虚拟机的代表作之一。它具备热点探测功能，可以通过此功能将一个被频繁调用的方法或者方法体中有效循环次数较多的代码块标记为“热点代码”，然后通过内嵌的双重 JIT（Just In Time Compiler）编译器将字节码直接编译为本地机器指令。在 HotSpot VM 内部，即时编译器与解释器是并存的，通过编译器与解释器的协同工作，既可以保证程序的响应时间，同时还能够提高程序的执行性能，并且对编译器的工作压力也降低了一定程度的负载。换句话说 HotSpot 是一款解释器与编译器并存的虚拟机，缺省情况下一个程序中到底有多少字节码指令是通过解释运行的，还是通过编译运行的，这就需要依赖热点探测功能。

虽然 Java 虚拟机规范并没有强制要求虚拟机内部实现一定要采用解释器和编译器并存的架构方案，但目前市面上大多数主流虚拟机都采用此架构。这是因为当虚拟机启动的时候，解释器可以首先发挥作用，而不必等待编译器全部编译完成再执行，这样可以省去许多不必要的编译时间。并且随着程序运行时间的推移，编译器逐渐发挥作用，根据热点探测功能，将有价值的字节码编译为本地机器指令，以换取更高的程序执行效率。HotSpot VM 中内嵌有两个 JIT 编译器，分别为 Client Compiler 和 Server Compiler，但大多数情况下我们简称为 C1 编译器和 C2 编译器。开发人员可以通过命令显式指定到底使用哪一种编译器策略，缺省情况下 HotSpot 会根据操作系统版本与物理机器的硬件性能进行自动选择。当然开发人员还可以通过命令显式指定 HotSpot VM 到底是使用完全编译策略，还是完全解释策略，如果我们将虚拟机选定为完全解释策略，那么编译器将停止所有的工作，字节码将完全依靠解释器逐行解释执行。反之也可以选用完全编译策略，但解释器仍然会在编译器无法进行的特殊情况下介入执行，这主要是确保程序能够最终顺利执行，如图 1-4 所示。

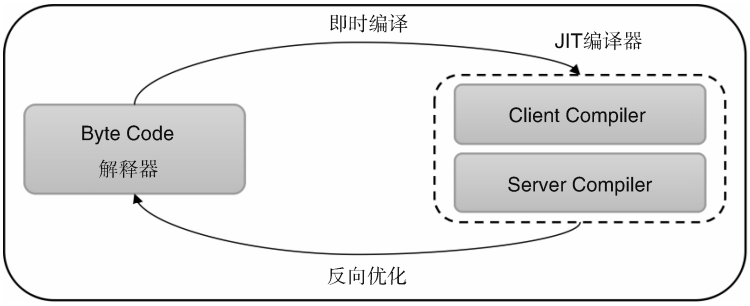


图 1-4 解释器与 JIT 编译器协作

Java7 缺省开启分层编译（Tiered Compilation）策略，由 C1 编译器和 C2 编译器相互协作共同来执行编译任务。C1 编译器会对字节码进行简单和可靠的优化，以达到更快的编译速度；而 C2 编译器会启动一些编译耗时更长的优化，以获取更好的编译质量。

其实除了 HotSpot 之外，目前市面上也不乏一些同样优秀的 Java 虚拟机产品，比如 IBM 公司研发的 IBM J9 VM，以及 Oracle 收购的 BEA JRockit VM 等。Oracle 公司将会在后续版本中，将 HotSpot VM 和 JRockit VM 合二为一，在 HotSpot VM 的基础之上整合 JRockit VM 的诸多优点。

1.3 安装与配置Java运行环境

所谓工欲善其事，必先利其器。在开始正式讲解后续章节之前，我们首先需要搭建好 Java 平台所需的一系列开发环境和运行环境。Java 的运行环境工具我们称之为 JDK（Java Development Kit，Java 软件开发工具包）。JDK 是 Oracle 公司提供的用于支持 Java 程序运行的开发工具包。该包中包含 Java 运行环境、Java 工具和 Java API。大家可以从 Oracle 的官方站点下载 JDK 工具包，本书所有程序示例均使用 jdk7-u15-x86 版本，建议大家还是下载和本书保持一致的版本，尽可能避免因版本问题而导致出现一些不一致的情况。

1.3.1 Windows环境下的安装与配置

当成功下载 JDK 后，大家可以选择默认与自定义风格两种模式来进行 JDK 的安装。当成功安装好后，我们可以发现在安装目录下会出现一个“java”文件夹。该文件夹内部还包含一个“jdk”文件夹与“jre”文件夹。“jre”文件夹中仅仅只是包含 Java API 组件与 Java 运行环境，而“jdk”文件夹中除了包含除 Java API 组件与运行环境外，还一些 Java 工具及 Java API 源码。

在 jdk 或 jre 文件夹的 bin 目录下，包含 JDK 最主要的两个工具，分别是 Java 虚拟机和

Java 前端编译器。Java 虚拟机叫做“Java.exe”，而 Java 前端编译器则叫做“Javac.exe”。当然还包含一些其他工具，本书会在后续章节中陆续为大家进行讲解其使用过程。

当成功安装好 JDK 后，我们还需要为其配置环境变量。配置环境变量的目的其实是在于引导运行环境正确执行 Java 应用程序。当然如果你电脑里安装有 IDE（集成开发环境）工具，理论上不需要配置 JDK，但如果需要将 Java 程序打包在本地运行时，Java 运行环境就显得至关重要了。并且对于 Eclipse 等开源的 IDE 工具而言，本身并无内置 JDK，所以笔者还是建议大家下载好 JDK 后，按照本书的步骤配置好 Java 环境变量，这样便能一劳永逸。

在 Windows 环境下配置 Java 环境变量比较简单，选择“系统属性->环境变量”，新增如下 3 个环境变量参数即可：

- JAVA_HOME=JDK 安装路径
- PATH=JDK 安装路径\bin 路径
- CLASS_PATH=.

当配置好环境变量后，我们便可以通过打开 Windows 控制台输入命令“javac”来查看 Java 环境变量是否成功配置，如果输入该命令后返回的是用法信息，则意味着 Java 的环境变量已经配置成功。大家还可以通过命令“java -version”查看当前安装的 JDK 版本。

1.3.2 Linux环境下的安装与配置

1.3.1 节讲解了关于如何在 Windows 环境下安装与配置 Java 运行环境。但在实际的程序开发过程中，我们往往需要将编写好的 Java 程序部署在 Linux 平台上运行。那么这个时候必然需要在 Linux 操作系统上安装与配置 Java 的一系列运行环境。其实和 Windows 平台相比起来，Linux 环境下的 Java 安装与配置则显得稍微繁琐一些。

在开始往 Linux 环境下安装与配置 Java 运行环境之前，大家首先需要确保自己的电脑里已经成功安装 Linux 操作系统。当然如果你没有多余的电脑进行 Linux 环境的搭建，那么可以在本机安装一个虚拟机工具来搭建 Linux 环境。由于目前市面上拥有较多开源的 Linux 操作系统版本，可能会导致某些刚接触 Linux 的开发人员无从选择，所以在此笔者推荐其操作系统版本为 ubuntu-12.04.1-x86/x64，因为该版本界面较为友好和人性化，适合刚接触 Linux 操作系统的开发人员。

当一切的基础环境都准备好以后，你可以登录 Oracle 的官方站点下载免安装版本的 JDK 工具包，或者通过打开 Linux 平台下的 shell 命名解析器输入命令“sudo apt-get install 名称”进行 JDK 的下载。建议选择的 Linux 免安装版本为 jdk7-u15-linux-x86。当成功下载好 JDK 后，我们需要将其解压，并存放于所指定的目录即可完成在 Linux 操作系统下的 Java 运行

环境安装。

当成功安装 Java 运行环境后，我们仍然需要配置 Java 的环境变量。打开 Linux 平台下的 shell 命名解析器输入命令“vi \etc\profile”对 profile 文件进行编辑，在最后一行新增如下 3 个环境变量参数：

- JAVA_HOME=JDK 安装路径
- PATH=JDK 安装路径\bin 路径
- CLASS_PATH=.

当成功配置好环境变量参数并保存退出后，大家还需要执行命令“source /etc/profile”使配置生效。这时我们便可以通过打开 Linux 平台下的 shell 命名解析器输入命令“javac”查看 Java 环境变量是否成功配置，如果输入该命令后返回的是用法信息，则意味着 Java 的环境变量已经配置成功。大家还可以通过命令“java -version”查看当前安装的 JDK 版本。

几乎所有的 Linux 操作系统预先就安装 Java 运行环境（OpenJDK），如果程序并没有特定依赖指定的 JDK 版本，还是建议大家不必重复安装。当然如果你仅仅只是感兴趣而想尝试着在 Linux 下安装 Java 环境变量倒也无妨。

1.3.3 编写Java程序

当一切环境与准备工作都做好后，接下来本书将带领大家编写第一个 Java 应用程序。这个程序非常简单，只需要在控制台输出一段字符即可，并不包含过多的 Java 语言语法特性。

打开你的记事本或者 Editplus 工具，把下述代码拷贝后粘贴进去，保存后更改文件的后缀名为“.java”。在此需要提醒大家，文件名称务必保持和 Java 类型名称一致，否则 Javac 编译器无法将你所指定的 Java 源代码成功编译为有效的字节码文件。如下所示：

代码 1-1 Java 程序示例

```
/**
 * 一个简单的 Java 程序
 *
 * @author JohnGao
 */
public class HelloWorld {
  public static void main(String[] args) {
      System.out.println("Hello World");
  }
}
```

上述程序示例中，main 函数作为 Java 程序的入口点。不管其处于代码的任意位置，Java 程序都会以 main 函数作为调用函数。函数的意思为某一特定块代码块所要执行的具体任务，只不过在 Java 语言中其实应该准确的称呼为“方法”。

System 是 Java API 的预定义类，主要是给开发人员提供对底层操作系统 API 的只读访问操作，比如开发人员可以使用 System 类型的 getProperty（"os.name"）函数获取当前操作系统的系统名称。当然上述程序示例中字符串的输出其实是由 PrintStream 类型的 println 函数完成的，而 System.out 实际上就是调用的系统内置输出流 PrintStream 类型。

1.3.4 编译与运行

当大家成功将 1.3.3 节的程序示例拷贝粘贴至记事本，并成功保存更改后缀名为“.java”后，则可以使用 JavaC 编译器对其进行字节码编译。当然在编译前务必需要确定本地环境变量是否成功配置，如果没有配置或者配置错误 Java 环境变量后则无法正确编译字节码。大家打开 Windows 的控制台，切换到 Java 源代码所在的路径后，输入如下编译语法即可成功将 Java 源代码编译成字节码文件：

```
“路径：\>javac 包名\类名.java”
```

当成功将 Java 源代码编译成字节码后，JavaC 编译器会在与 Java 源代码的同级目录生成一个与源代码名称一致但后缀为“.class”的字节码文件。大家打开 Windows 控制台，切换到 Java 字节码所在的路径后，输入如下语法即可成功执行 Java 程序：

```
“路径：\>javac 包名\类名.java”
```

当成功编译并运行上述 Java 程序后，控制台将会输出如下字符信息：

```
Hello World
```

1.3.5 关键字与标示符

Java 目前大约定义了 51 个关键字。其实所谓关键字指的就是 Java API 内部预定义的一些字符集合（如 public、private、class 等），这些关键字与 Java 语法结构息息相关，同样开发人员也可以称其为保留字。至于标示符其实就是用于给类、方法、变量命名，只要在程序中可以自己命名的地方，统统都可以称之为标示符。在此需要提醒大家，Java 语法结构规定开发人员不允许在程序中使用关键字作为标示符命名，这一点请大家一定要牢记，否则将无法通过编译。

其实不仅仅是 Java，任何程序设计语言对于标示符的命名都遵循了业界统一的命名规范。虽然命名规范并不强制要求开发人员一定要遵循，但这对于程序开发风格的统一将会有

很大好处。如果你希望你编写的代码拥有更好的可读性和维护性，那么就应该遵循这些习惯。

对于类型标示符的命名，开发人员应该按照首字母大写的方式进行标示，比如“Xxxx”。而变量、方法则应该按照“驼形命名规则”的方式进行标示，也就是首字母需要小写，比如“xXxxX”。而对于包的命名，开发人员则需要将所有字母小写，比如“com.xxx.xxx”。常量的命名则最为特殊，所有的字母均全部按照大写的方式进行标示，比如“XXXX”。当然如果常量由多个单词组成，这时为了确保可读性，开发人员可以使用“_”符号进行分割。

其实掌握命名规范并不困难，因为这仅仅只是一种习惯，但还是需要提醒刚接触编程的开发人员，请务必遵循如下守则：

- □ 不要采用中文编码的方式对标示符进行命名，务必只使用英文字符；
- □ 不要使用特殊符号的方式对标示符进行命名，比如“@#￥%……&*（）-+=”，但允许使用符号“_”，但建议尽量不要标示在首字母第一位；
- □ 如果需要以数字的方式对标示符进行命名，不能标注在首字母第一位；
- □ 标示符不能以关键字进行命名。

一般来说程序开发所选用的字符语言都是英文字母，这是从编程语言诞生至今就养成的一种根生蒂固的习惯。但不是说不能使用其他字符进行程序开发，前提是你需要确认程序的开发工作是一个团队还仅有你一个人，可读性才是需要考虑的首要问题。

目前国内的确已经出现一种采用中文字符进行编程的程序设计语言（API 与关键字都设定为中文字符）。该设计者的设计初衷确实能够有效解决某一部分对英文“过敏”的开发人员的读写问题，但是对于大部分本土开发人员而言，它始终受众面过于狭隘，导致该语言的推广力度较为艰难。

1.4 Java技术的新特性

2005 年 6 月 28 日，由全球一万多名开发人员出席的 Java One 开发者大会上，Sun 公司老板 Scott McNealy 先生向 Java 之父 James Gosling 博士颁发了“终身成就奖”，这不仅是 Sun 公司对 James Gosling 博士十多年来创新性工作的肯定和感谢，同时也意味着 Java 已经成为世界上应用最为广泛的技术之一。作为 Java One 开发者大会上的压轴戏，James Gosling 博士做了一场命题为“Java 技术下一个十年”的演讲。

1.4.1 Java模块化与OSGi技术

近几年来，Java 模块化一直是一个比较活跃的话题。那么究竟什么是模块化呢？其实

所谓模块化指的就是开发人员在构建大型系统时，能够将系统中的每一个功能模块进行独立的开发和物理部署，这样做的优点不仅能够有效降低各个业务模块之间的耦合，同时还能够保证当单一模块发生故障的时候不会影响系统整体的运行。当然模块化本身只是一种概念，其目的就是为了将系统中原本耦合的逻辑进行分解，以此满足各个模块之间的独立，并定义一种标准化的接口契约来进行相互之间的通信。

尽管 Java 目前并没有在 JDK 中内置模块化编程技术，但这似乎并不能阻挡开发人员选用 OSGi 技术作为模块化编程的首选。早在 2007 年的时候，由 Sun 公司主导并提交的 JSR-277（Java 模块化系统）规范并没有通过 JCP 组织的审核，这主要是由于 JCP 专家组织通过投票将 IBM 公司提交的 JSR-291（OSGi R4.1）纳入了 Java 模块化规范标准。直到 Sun 公司在 Java7 早期时，再次提交 JSR-294（Java 模块化系统的改进支持）规范，可惜最终还是未能如愿。不得已 Sun 公司只能够避开 JCP 组织，在 OpenJDK 中创建了一个叫作 Jigsaw 的子项目来实现 Java 模块化编程技术，但该项目却被迫延期到 Java 9 中进行发布。我们先不论 Jigsaw 能否顺利在 Java 9 中出现，就目前而言 OSGi 技术早已是 Java 模块化规范标准，不得不承认 Sun 公司在这一场模块化规范争夺战中是以失败告终的。或许在后续的 Java 版本中，我们能够看到 Jigsaw 与 OSGi 技术的整合，共同为 Java 的模块化编程带来新的体验。

1.4.2 语言无关性

在很多年以前，想要能够在 Java 虚拟机平台上运行非 Java 语言编写的程序，这简直就是天方夜谭，但随着 Java7 的正式发布，这已经不再是一个奢侈而遥远的梦想，Java 虚拟机的设计者们通过 JSR-292 规范基本兑现了这个承诺。随着软件开发领域的复杂度日渐增加，使得开发人员使用单一的 Java 语言进行项目开发已经显得有些力不从心，如果 Java 语言能够与其他编程语言进行混合编程，不但能够弥补自身不足，同时又能够利用其他编程语言的优点很好地解决技术问题，这无疑是一件美好的事情。也就是说某些 Java 语法层面并不支持的特性，并不代表 Java 虚拟机也无法支持，只要其他编程语言能够有效支持，且编译结果是有效的字节码文件，Java 虚拟机就能够将其装载进内部并顺利运行，这就是为什么需要使用混合语言编程的目的和好处。

本书在此所提及的混合编程概念，指的是在同一个虚拟机的宿主环境下，能够同时运行使用 Java 语言以及其他编程语言编写的应用程序。这一切听起来似乎很神奇，Java 虚拟机为什么能够解释出非 Java 语言编写的程序呢？其实 Java 虚拟机根本不关心运行在其内部的程序到底是使用何种编程语言编写的，它只关心“字节码”文件。也就是说 Java 虚拟机拥有语言无关性，并不会单纯地与 Java 语言“终身绑定”，只要其他编程语言的编译结果满足并包含 Java 虚拟机的内部指令集、符号表以及其他的辅助信息，它就是一个有效的字节码文件，就能够被虚拟机所识别并装载运行，如图 1-5 所示。

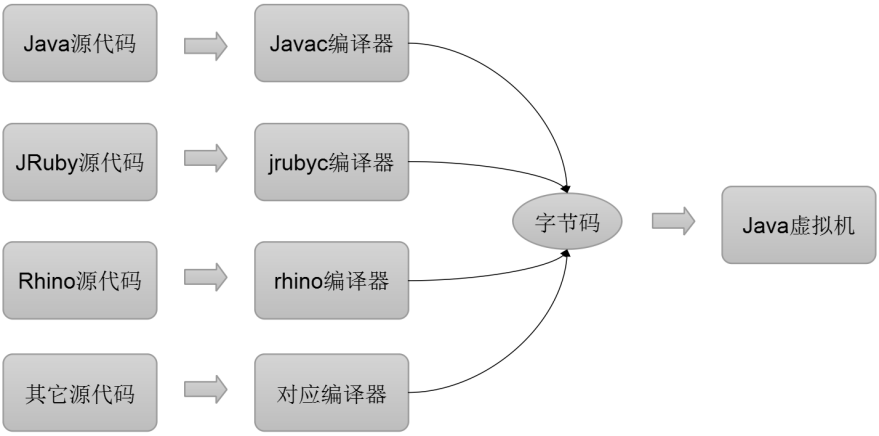


图 1-5 Java 虚拟机的与语言无关特性

如果任何编程语言的编译结果都可以满足字节码的组成结构和存储形式，那么从理论上来讲都可以运行在 Java 虚拟机内部。在此大家需要注意，本书所提及的任何编程语言如果希望编译结果是字节码，那么必然需要对应 Java 虚拟机实现的语言版本。比如大家熟知的：Ruby（JRuby）、JavaScript（Rhino）、PHP（IBM WebSphere sMash PHP（P8）、Caucho Quercus）等优秀的编程语言都具有其对应的虚拟机实现版本。

1.4.3 使用Fork/Join框架实现多核并行

随着如今硬件水平的高速发展和提升，就连手机等智能移动设备的 CPU 核心都采用多核超线程技术，而传统的 PC 设备、企业级服务器自然也由早期关注的高频率转换为多核心去处理日常任务。在今天如果一门编程语言不能高效地支持多核并行计算，无论其曾经多么优秀和辉煌，都将注定被开发人员所淘汰，毕竟这是谁也无法阻止的优胜劣汰原则。Java 早在多核时代还没来临的时候，就已经开始支持单核并发计算，但随着多核设备的日渐普及，开发人员更希望 Java 能够充分利用所有可用物理核心一起高效地并行处理任务。所以在 Java5 版本的时候，Java 设计者们通过 JSR-166 的规范制定，在 java.util.concurrent 包下为开发人员提供了基于粗粒度的多核并行计算框架，只不过这种基于粗粒度的并行计算模式，并不能在处理效率上达到令人满意的程度。因为这种基于粗粒度的并行计算，根本无法高效组合所有可用物理核心一起进行并行任务处理，甚至在某些情况下，还有可能导致部分物理核心处于空闲等待状态。

由于粗粒度的并行计算并不能够充分挖掘多核处理器的性能，所以 Java 设计者们又对 JSR-166 规范进行了一系列的整改修订，并在 Java7 版本中在 java.util. concurrent.forkjoin 包下新增了基于细粒度的多核并行计算 Fork/Join 框架。该框架的设计初衷是将一个任务量化到最小，并提供高计算密度的并行处理性能。简单来说，我们可以将一个任务拆分成若干个

子任务，直到这个任务足够小，然后每一个子任务被独立并行计算，直到任务执行完成，再将其逐个合并为一个完整的任务，这就是 Fork/Join 框架提供的细粒度并行计算模式，如图 1-6 所示。

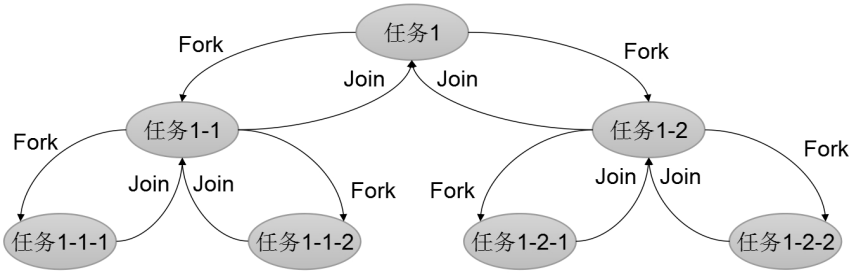


图 1-6 Fork/Join 框架计算模型

除了Java7 能够高效地利用Fork/Join框架实现多核并行计算外，开源基金会Apache提供的Hadoop[⑤] Map/Reduce框架也是一个高效的海量数据计算框架，它允许开发人员将其部署在廉价的集群服务器上处理TB级别的数据。当然还有一些编程语言自诞生那天起，就是为了解决并行计算而来，比如Scala、Clojure和Erlang等。这类编程语言，不仅继承了面向对象（Object Oriented）的特性，同时还结合了函数式编程等特性。虽然目前Java同样也能够使用函数式编程，但代码将会显得非常冗余，不过Java设计者们在Java8 中提供对Lambda的支持，这会极大改善Java对于函数式编程的不足。

1.4.4 丰富的语法特性

Java7 的新特性更多的是体现在语法层面上的扩充，这对于 Java 语法的易用性和准确性来说将会有非常大的提升。比如 switch 表达式将提供对 String 类型的支持、直接二进制字面值的定义、try-with-resources 实现自动资源管理、cache 语句表达式的改变、泛型的“<>”类型推断运算符支持、全新的文件系统 NIO2.0、Fork/Join 模型等。

Java7 最激动人心的地方无疑是全新的文件系统 NIO2.0 的到来。利用 NIO2.0，开发人员则无需关注 I/O 细节，因为新文件系统中封装有大量的通用操作，便于开发人员更好地关

⑤ Hadoop 是一个分布式系统基础架构，由 Apache 基金会开发。用户可以在不了解分布式底层细节的情况下开发分布式程序，充分利用集群的威力高速运算和存储。Hadoop 实现了一个分布式文件系统（Hadoop Distributed File System，简称 HDFS）。HDFS 有着高容错性的特点，并且设计用来部署在低廉的（low-cost）硬件上。而且它提供高传输率（high throughput）来访问应用程序的数据，适合那些有着超大数据集（large data set）的应用程序。Hadoop 的组成结构比较复杂，它由许多元素构成。其最底部是 HDFS，它存储 Hadoop 集群中所有存储节点上的文件。HDFS（对于本文）的上一层是 MapReduce 引擎，该引擎则由 JobTrackers 和 TaskTrackers 组成。

注自身业务。并且在 I/O 模型方面，将支持调用操作系统的 IOCP（Input/Ouput Completion Port，输入/输出完成端口）接口实现真正的异步 I/O，尽可能避免因 I/O 问题导致的系统瓶颈出现。

唯一遗憾的就是原本归纳在 Java7 中的 Lambda 表达式，却被迫到 Java8 中进行发布。不过当 Lambda 表达式真正到来时，也意味着 Java 将会走向函数式编程的道路，这对 Java 语法和开发人员的编码习惯都会带来巨大的影响，毕竟函数式编程在未来的几年内很可能成为主流。

1.4.5 过渡到 64 位虚拟机

早在市面上第一款 64 位处理器诞生不久后，Sun 公司随即推出了用于支持 64 位系统的 JDK。相对于传统的 32 位虚拟机，64 位虚拟机所具备的最大优势就是可以访问大内存，相信大家都知道 32 位虚拟机的最大可用内存空间被限定在了 4GB，并且 Java 堆区的大小如果是在 Windows 平台下最大只能设置到 1.5GB，而在 Linux 平台下最大也只能设置到 2GB～3GB 的上限，也就是说，Java 堆区的内存大小设置还需要依赖于具体的操作平台。既然 32 位虚拟机无法满足大内存消耗的应用场景，那么 64 位虚拟机的出现则是顺理成章，64 位虚拟机之所以能够访问大内存，是因为其采用了 64 位的指针架构，这也是寻址访问大内存的关键要素。

在 JDK1.6 Update14 版本之前，64 位虚拟机的综合性能表现实际上是不如 32 位虚拟机的，这主要是因为 OOPS（Ordinary Object Pointers，普通对象指针）从 32 位膨胀到 64 位后，CPU Cache Line 中的可用 OOPS 变少，这样一来将会直接影响并降低 CPU 的缓存使用率，这就是 64 位虚拟机在性能上之所以落后于 32 位虚拟机的主要原因。其次由于部署在 64 位虚拟机中的程序大部分都需要用到大内存，尤其是互联网项目，经常需要使用多达几十乃至上百 GB 的内存，这对于传统的 32 位虚拟机将无法承载（部分企业采用 32 位虚拟机集群的方式使用大内存），只能依靠 64 位虚拟机去支撑。但是管理这么大的内存开销对于 GC 来说将会是一场非常严峻的考验，甚至很有可能会导致 GC 在执行内存回收期间消耗更长的时间，同时也就意味着工作线程的等待时间将会延长。随着如今 64 位虚拟机的逐渐成熟，JVM 的设计者们在 JDK1.6 Update14 版本开始提供了指针压缩功能，也就是说，指针压缩将会通过对齐补白等操作将 64 位指针压缩为 32 位，以此改善 CPU 缓存使用率达到提升 64 位虚拟机运行性能的目的。

在此需要提醒大家，如果所使用的 64 位虚拟机的版本在 Update14-Update22 之间，可以通过选项“-XX:+UseCompressedOops”显式开启指针压缩功能，而在 Update23 版本之后，指针压缩功能将会被缺省开启。

1.5 实战：玩转OpenJDK

谈到 OpenJDK 相信大家并不会感觉到陌生，在 CentOS、Ubuntu 等常用的 Linux 发行版操作系统上，几乎都预装 OpenJDK 作为缺省的 Java 运行环境。一般来说，使用 OpenJDK 已经完全可以满足大多数的应用场景，笔者公司的项目就是直接部署在 OpenJDK 中运行。当然如果你想要彻底弄清楚 JDK 的内部实现原理，那么亲自动手编译一套 JDK 以及 Debug HotSpot 则是最直接的一种方式。千万不要认为编译 OpenJDK 需要多么深厚的技术功底，实际上这是非常简单的，只需要开发人员掌握最基本的 shell 命令，就可以成功编译出一套自己的 JDK。

在 OpenJDK 中，除了 HotSpot 是使用 C++以及混合了少量的 C 和汇编语言编写外，大部分内容其实都是使用 Java 语言编写的，比如 Java 的基础类库等。目前市面上开源的众多 JDK 中，开发人可以选择 Apache Harmony（该项目已于 2011 年 10 月宣布停止开发）、OpenJDK 等进行编译。考虑到 OpenJDK 的应用最为广泛，所以本书就选择使用 OpenJDK 作为这次编译实战的开源 JDK 版本。

1.5.1 JDK与OpenJDK的关系

OpenJDK简单来说就是Sun/Oracle JDK的一种开源版本，但OpenJDK却并不属于Sun和Oracle等商业公司，它属于开源社区，任何组织或个人都可以为推动Java未来的技术发展做出贡献。Sun公司早在 2006 年的时候就宣布会将JDK以GPL v2 的开源协议进行源代码公开，但直到 2009 年Sun公司才正式发布第一个开源的JDK版本，那就是如今的OpenJDK。目前开发人员使用的OpenJDK7 与Oracle JDK7 中的代码几乎是一模一样的，唯一的区别就是Oracle JDK7 中的部分代码因为版权问题在OpenJDK7 中只能使用其他的技术进行替代，除此之外均没有任何区别，如图 1-7 所示 ⑥。

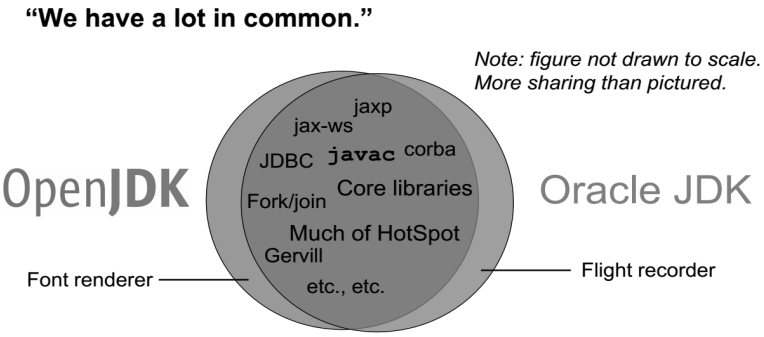


图 1-7 OpenJDK 与 Oracle JDK 的关系

⑥ 图片来源于：https://blogs.oracle.com/darcy/resource/OSCON/oscon2011_OpenJDKState.pdf 文档中。

和 OpenJDK7 不同，OpenJDK6 和 Sun JDK6 的源代码则存在较大差异，因为 OpenJDK6 仅仅只是 OpenJDK7 的一个分支，为了尽量符合 Java6 的标准，OpenJDK6 中删除了所有关于 Java7 的新特性。并且 OpenJDK 和 Sun/Oracle JDK 的源代码开源协议也不同，OpenJDK 采用的是 GPL V2 协议，而 Sun/Oracle JDK 采用的则是 JRL（JavaResearch License）协议。采用 GPL V2 协议发布的 OpenJDK 源代码允许商业用途，而采用 JRL 协议发布的 Sun/Oracle JDK 源代码则仅限于个人研究使用。除此之外 OpenJDK 中并不会包含 Deployment 功能，比如 Browser Plugin、Java WebStart 和 Java 控制面板等，因此开发人员无法在 OpenJDK 中使用这些功能。除了无法使用 Deployment 功能外，OpenJDK 中也不会包含 Rhino 和 Java DB 等工具，也就是说，OpenJDK 其实只是一个精简版的 JDK 而已，由于 OpenJDK 的不完整性，因此被认定为是非标准版的 JDK 版本，所以也就无法正常使用 Java 商标。但如果使用 Icedtea 补丁的 OpenJDK 版本，当输入命令“java -version”后，则会显示 Java 版本号，而非 OpenJDK 版本号。

尽管 OpenJDK 在某些方面和 Sun/Oracle JDK 还存在一定差异，但已经完全可以满足大多数的应用场景，并且 OpenJDK 还为开发人员研究 JDK 的内部实现原理提供了便捷。

1.5.2 基于OpenJDK深度定制的淘宝JVM（TaobaoVM）

使用 Java 技术编写的系统，无疑在生产环境中需要对 Java 虚拟机进行正常的调优工作，既然谈到 Java 虚拟机的调优技术，笔者相信大部分的开发人员至今仍然仅停留在参数调制上。由于淘宝目前无疑是中国最大的 Java 技术应用方，那么淘宝究竟是采用什么样的技术对 Java 虚拟机进行优化的呢？淘宝的技术团队对 Java 虚拟机的优化工作其实早已不是停留在简单的参数调制上面，而是充分结合了企业自身的业务特点以及实际的应用场景，在 OpenJDK 的基础之上通过修改大量的 HotSpot 源代码，深度定制了淘宝专属的高性能 Java 虚拟机——TaobaoVM。

既然是结合业务特点深度定制的一款 Java 虚拟机，那么性能必然在某一些特定的应用场景上会比 Oracle 官方的 HotSpot 更强，如图 1-8 所示。但其弊端同样也非常明显，那就是无法实现通用。所以如果只是想对 TaobaoVM 进行研究的话，可以参考 jvm.taobao.org 中的描述编译一个 TaobaoVM，但如果需要应用在实际的项目中，笔者还是建议三思而后行，否则将会得不偿失。

淘宝的技术团队通过修改大量的HotSpot源代码深度定制的TaobaoVM⑦，其实从严格意义上来说，在提升Java虚拟机性能的同时，却严重依赖物理CPU类型。也就是说，部署有TaobaoVM的服务器中，CPU全都是清一色的Intel CPU，且编译手段采用的是Intel C/CPP

⑦ 文章完整地址：http://os.51cto.com/exp/velocity2012/ppt/wangcheng.pdf。

Compiler进行编译，以此对GC性能进行提升。除了优化编译效果外，TaobaoVM还使用了crc32指令实现JVM intrinsic降低JNI的调用开销，如图 1-9 所示。

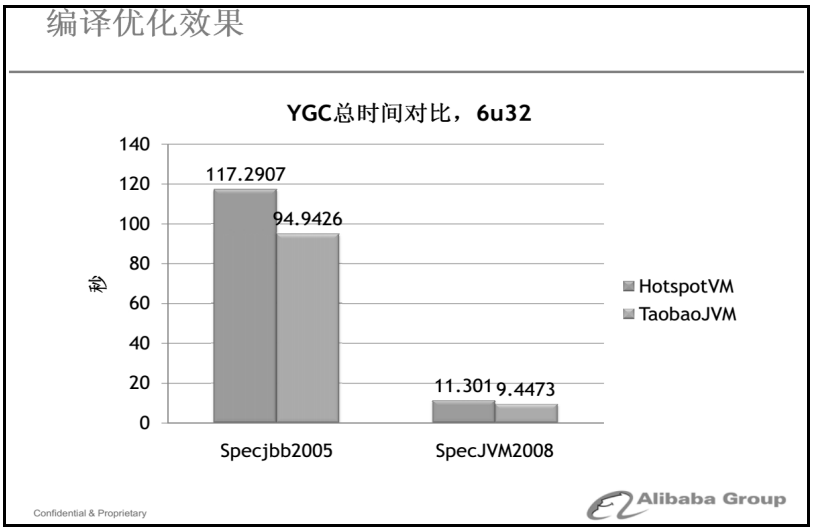


图 1-8　TaobaoVM 的优化编译效果

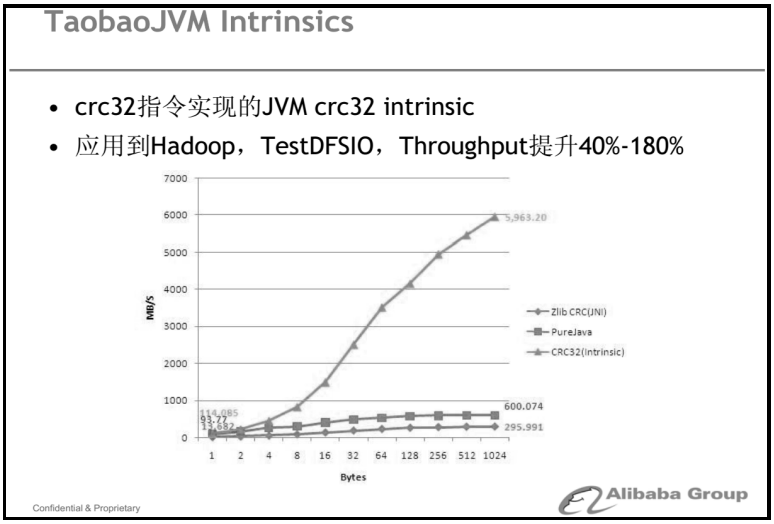


图 1-9　JVM intrinsic 降低 JNI 的调用开销

除了在性能优化方面下足了功夫，TaobaoVM 还在 HotSpot 的基础之上大幅度扩充了一些特定的增强实现。比如创新的 GCIH（GC invisible heap）技术实现 off-heap，这样一来就可以将生命周期较长的 Java 对象从 heap 中移至 heap 之外，并且 GC 不能管理 GCIH 内部的 Java 对象，这样做最大的好处就是降低了 GC 的回收平率以及提升了 GC 的回收效率，并且

GCIH 中的对象还能够在多个 Java 虚拟机进程中实现共享。其他扩充技术还有利用 PMU hardware 的 Java profiling tool 和诊断协助功能等。

1.5.3 下载OpenJDK源代码

大家可以登录 http://openjdk.java.net/下载 OpenJDK 的源代码，本书编译实战所演示的 JDK 版本为 openjdk7-u40。当然你也可以选择下载其他版本的 OpenJDK，但是为了避免在编译过程中发生一些不必要的错误，所以建议大家尽量保持和本书一致的 JDK 版本。

想要下载 OpenJDK 的源代码一共有两种方式。一种是直接使用 Mercurial 版本管理工具从 Repository 中 check out 出源代码（地址为：http://hg.openjdk.java.net/jdk7u/jdk7u40）。但由于绝大多数的 Java 开发人员或许并不熟悉 Mercurial 工具的使用，以及从 Repository 中 check out 出源代码耗时太久，所以除了采用这种方式外，大家还可以采用手动下载的方式从 http://download.java.net/openjdk/jdk7u40 下载 OpenJDK 的源代码。

当成功下载好 OpenJDK 的 zip 包并使用命令“unzip”将其进行解压后，便可以在解压后的 OpenJDK 根目录中找到 README-builds.html。该文档中详细地描述了 OpenJDK 的完整编译步骤，所以如果你是第一次编译 OpenJDK，那么阅读该文档则会事半功倍。

1.5.4 构建编译环境

前面提到过，如果想要彻底弄清楚 JDK 的内部实现原理，亲自动手编译一套 JDK 及 Debug HotSpot 则是最直接的一种方式。但是在执行编译之前，构建编译环境则显得至关重要，因为编译环境直接决定了编译结果，如果没有构建出完整的编译环境则无法成功编译 JDK。

大家几乎可以在任何的系统平台上搭建 OpenJDK 的编译环境，比如 Linux 平台、Mac OS 平台甚至是 Window 平台上都可以构建编译环境。只不过想要在 Window 平台上构建编译环境则显得非常麻烦，而且还容易出错。因为 Window 平台不仅需要依赖 Cygwin 等虚拟机环境，更重要的是无法确保构建环境期间的正确性，所以本书选择在 Linux 平台上构建编译环境，相对 Windows 来说不仅构建成本低廉，还更简单和直接。

本书选用的 Linux 发行版为 ubuntu-12.04.1-x86，其 Linux 内核版本为 3.2。当成功准备好系统平台后，接下来的任务就是构建编译环境。想要成功编译出 JDK，还需要下载和安装 Bootstrap JDK、GCC、Make 和 Ant 等一系列的项目依赖。在 OpenJDK 中除了 HotSpot 是使用 C++以及混合了少量的 C 和汇编语言编写外，大部分内容其实都是使用 Java 语言编写的，比如 Java 的基础类库等，所以这时就需要用到用于编译 Java 代码的 JDK，我们可以将其称之为 Bootstrap JDK。而 C/C++编写的代码则需要使用 GCC 编译器进行编译，在 Ubunto

中 GCC 缺省就已经安装了。当 OpenJDK 中的所有源代码都已经成功编译后，剩下的任务就是 Build 操作。使用 Bootstrap JDK 编译的 Java 代码需要使用 Ant 工具进行 Build，而使用 GCC 编译的 C/C++代码则需要使用 Make 工具进行 Build。

本书编译实战所使用的项目依赖版本如下：

- Bootstrap JDK 版本：openjdk6-b31；
- GCC 编译器版本为：4.6.3；
- Make 工具版本为：3.8.1；
- Ant 工具版本为：1.8.2。

如果不想手动下载编译 JDK 所需的一系列项目依赖，则可以使用如下命令一次性下载和安装完成，如下所示：

```
sudo apt-get install build-essential gawk m4 openjdk-6-jdk
libasound2-dev libcups2-dev libxrender-dev xorg-dev xutils-dev
X11proto-print-dev binutils libmotif3 libmotif-dev ant
```

当成功执行上述命令后，则会自动下载和安装编译 OpenJDK 所需的一系列项目依赖，这样就可以将大家从繁琐的手动操作中解放出来，全身心地投入到编译任务中去。

1.5.5 执行整个OpenJDK的编译

当成功构建好 OpenJDK 所需的编译环境后，最后还需要设置一些环境变量参数，因为在编译 OpenJDK 的时候，需要用到这些环境变量做引导。参考 README-builds.html 文档中有关环境变量的设置，其中只有 LANG 和 ALT_BOOTDIR 这两个环境变量必须手动设置外，其他环境变量参数在编译 OpenJDK 时均可以使用默认设置，如下所示：

代码 1-2 设置编译 OpenJDK 所需的环境变量参数

```
#设定语言选项
export LANG=C

#定义 Bootstrap JDK 的路径
export ALT_BOOTDIR=/usr/lib/jvm/java-6-openjdk

#允许在编译过程中自动下载相关依赖

export ALLOW_DOWNLOADS=true

#并行编译策略时开启的线程数量
export HOST_BUILD_JOBS=4

#使用预编译头文件，加速编译
```

```
export USE_PRECOMPILED_HEADER=true

#编译内容，如果不设置，则编译 OpenJDK 中的所有内容，但耗时较长
export BUILD_LANGTOOLS=true
export BUILD_JAXP=true
export BUILD_JAXWS=true
export BUILD_CORBA=true
export BUILD_HOTSPOT=true
export BUILD_JDK=true

#禁止 build 出安装包
BUILD_INSTALL=false

#编译后的存储路径
export ALT_OUTPUTDIR=/home/johngao/openjdk7/build

#禁止编译器把一些警告当错误
WARNINGS_ARE_ERRORS=

#执行编译，并将编译过程中产生的日记进行存储
make 2>&1 | tee /home/johngao/openjdk7/build-log/openjdk.log
```

当成功设置好编译 OpenJDK 所需的环境变量后，为了避免每次都手动去执行，笔者建议大家在解压后的 OpenJDK 根目录中创建一个 shell 脚本，将上述内容粘贴进去。为了验证上述环境变量配置的正确性，可以使用命令“make sanity”进行检测，如下所示：

代码 1-3 使用 make sanity 检测

```
root@johngao-virtual-machine:/home/johngao/openjdk7/src/openjdk# ./build.sh
Build Machine Information:
   build machine = johngao-virtual-machine

Build Directory Structure:
   CWD = /home/johngao/openjdk7/src/openjdk
   TOPDIR = .
   LANGTOOLS_TOPDIR = ./langtools
   JAXP_TOPDIR = ./jaxp
   JAXWS_TOPDIR = ./jaxws
   CORBA_TOPDIR = ./corba
   HOTSPOT_TOPDIR = ./hotspot
   JDK_TOPDIR = ./jdk


Build Directives:
   BUILD_LANGTOOLS = true
   BUILD_JAXP = true
   BUILD_JAXWS = true
   BUILD_CORBA = true
```

```
    BUILD_HOTSPOT = true
    BUILD_JDK    = true
    DEBUG_CLASSFILES =
    DEBUG_BINARIES =

Hotspot Settings:
      HOTSPOT_BUILD_JOBS  =
      HOTSPOT_OUTPUTDIR  = /home/johngao/openjdk7/build/hotspot/outputdir
      HOTSPOT_EXPORT_PATH = /home/johngao/openjdk7/build/hotspot/import

Bootstrap Settings:
  BOOTDIR = /usr/lib/jvm/java-6-openjdk
    ALT_BOOTDIR = /usr/lib/jvm/java-6-openjdk
  BOOT_VER = 1.6.0 [requires at least 1.6]
  OUTPUTDIR = /home/johngao/openjdk7/build
    ALT_OUTPUTDIR = /home/johngao/openjdk7/build
  ABS_OUTPUTDIR = /home/johngao/openjdk7/build

—省去部分日志内容—

OpenJDK-specific settings:
  FREETYPE_HEADERS_PATH = /usr/include
    ALT_FREETYPE_HEADERS_PATH =
  FREETYPE_LIB_PATH = /usr/lib
    ALT_FREETYPE_LIB_PATH =

Previous JDK Settings:
  PREVIOUS_RELEASE_PATH = USING-PREVIOUS_RELEASE_IMAGE
    ALT_PREVIOUS_RELEASE_PATH =
  PREVIOUS_JDK_VERSION = 1.6.0
    ALT_PREVIOUS_JDK_VERSION =
  PREVIOUS_JDK_FILE =
    ALT_PREVIOUS_JDK_FILE =
  PREVIOUS_JRE_FILE =
    ALT_PREVIOUS_JRE_FILE =
  PREVIOUS_RELEASE_IMAGE = /usr/lib/jvm/java-6-openjdk
    ALT_PREVIOUS_RELEASE_IMAGE =

Sanity check passed.
```

输入命令“make sanity”后，如果输出日志最后提示“Sanity check passed.”，且没有任何警告信息时，则意味着环境变量配置正常可以执行 make 操作。在此需要提醒大家，全量编译整个 OpenJDK 时，编译过程会相当耗时，笔者全量编译大概耗费了一个小时左右的时间，所以大家只需在编译过程中耐心等待即可。当然如果只是增量编译，编译过程则会比较迅速。

当成功编译完 OpenJDK 后，如果输出日志最后提示如下信息则意味着你已经成功编译

好了一个完整的 OpenJDK，如下所示：

```
#-- Build times ----------
Target all_product_build
Start 2014-06-28 15:51:42
End   2014-06-28 16:59:30
00:01:57 corba
00:37:57 hotspot
00:00:35 jaxp
00:00:38 jaxws
00:25:36 jdk
00:01:03 langtools
01:07:48 TOTAL
-------------------------
```

从上述的输出日志中可以发现，笔者编译整个 OpenJDK 一共耗时 1 小时 07 分 48 秒，其中 HotSpot 和 JDK 的编译过程耗时最久，大概占了整个编译周期的 95%。当成功编译好 OpenJDK 后，大家可以通过环境变量 ALT_OUTPUTDIR 中配置的存储路径找到编译后的 OpenJDK 根目录，在根目录下的 j2sdk-image 目录中，所存放的内容就是编译后的 OpenJDK。

当我们输入命令“java -version”时，便可以查看由自己亲手编译的 OpenJDK 的版本信息，如下所示：

```
root@johngao-virtual-machine:/home/johngao/openjdk7/build/j2sdk-image/
  bin# ./java -version
openjdk version "1.7.0-internal"
OpenJDK Runtime Environment (build 1.7.0-internal-root_2014_06_28_15_51-b00)
OpenJDK Client VM (build 24.0-b56, mixed mode)
root@johngao-virtual-machine:/home/johngao/openjdk7/build/j2sdk-image/bin#
```

1.5.6 执行单独HotSpot的编译

其实编译 OpenJDK 真正有吸引力的地方是在 HotSpot 的编译部分，而非整个 OpenJDK，所以如果你只是想在成功编译好 HotSpot 后进行 Debug，则可以在 OpenJDK 源代码根目录下的/hotspot/make 目录中使用 Make 命令执行 Makefile 脚本即可。当然编译 HotSpot 和编译 OpenJDK 类似，都需要在编译时设置 LANG 和 ALT_BOOTDIR 这 2 个环境变量，如下所示：

代码 1-4 设置编译 HotSpot 所需的环境变量参数

```
#设定语言选项
export LANG=C

#定义 Bootstrap JDK 的路径
export ALT_BOOTDIR=/usr/lib/jvm/java-6-openjdk
```

```
#编译后的存储路径
export ALT_OUTPUTDIR=/home/johngao/openjdk7/build-jvm

make jvmg jvmg1 2>&1 | tee /home/johngao/openjdk7/build-log/hotspot.log
```

无论是编译整个 OpenJDK 还是只编译 HotSpot，需要设置的环境变量始终只有两个是必需的，所以笔者在编译 HotSpot 时，只设置了 LANG 和 ALT_BOOTDIR 以及指定编译后存储路径的 ALT_OUTPUTDIR 环境变量。

在/hotspot/make 目录下的 Makefile 脚本中，定义了 HotSpot 的编译目标类型，其中主要包括了 product、optimized、fastdebug 和 debug 等 4 种级别，如下所示：

代码 1-5　Makefile 脚本中 HotSpot 的编译目标类型

```
# Typical C1/C2 targets made available with this Makefile
C1_VM_TARGETS=product1 fastdebug1 optimized1 jvmg1
C2_VM_TARGETS=product  fastdebug  optimized  jvmg
ZERO_VM_TARGETS=productzero fastdebugzero optimizedzero jvmgzero
SHARK_VM_TARGETS=productshark fastdebugshark optimizedshark jvmgshark

COMMON_VM_PRODUCT_TARGETS=product product1 docs export_product
COMMON_VM_FASTDEBUG_TARGETS=fastdebug fastdebug1 docs export_fastdebug
COMMON_VM_DEBUG_TARGETS=jvmg jvmg1 docs export_debug

# JDK directory list
JDK_DIRS=bin include jre lib demo

all:          all_product all_fastdebug


ifdef BUILD_CLIENT_ONLY
all_product:   product1 docs export_product
all_fastdebug: fastdebug1 docs export_fastdebug
all_debug:     jvmg1 docs export_debug
else
ifeq ($(MACOSX_UNIVERSAL),true)
all_product:   universal_product
all_fastdebug: universal_fastdebug
all_debug:     universal_debug
else

all_product:   $(COMMON_VM_PRODUCT_TARGETS)
all_fastdebug: $(COMMON_VM_FASTDEBUG_TARGETS)
all_debug:     $(COMMON_VM_DEBUG_TARGETS)
endif
endif
```

```
all_optimized: optimized optimized1 docs export_optimized

allzero:          all_productzero all_fastdebugzero
all_productzero:   productzero docs export_product
all_fastdebugzero: fastdebugzero docs export_fastdebug
all_debugzero:     jvmgzero docs export_debug
all_optimizedzero: optimizedzero docs export_optimized

allshark:          all_productshark all_fastdebugshark
all_productshark:   productshark docs export_product
all_fastdebugshark: fastdebugshark docs export_fastdebug
all_debugshark:     jvmgshark docs export_debug
all_optimizedshark: optimizedshark docs export_optimized
```

由于编译后的 HotSpot 需要进行调试，所以笔者使用的 Make 命令为“make jvmg jvmg1”，这样一来就确定了 HotSpot 的编译目标类型是 debug 级别。

当成功编译好 HotSpot 后，大家可以通过环境变量 ALT_OUTPUTDIR 中配置的存储路径找到编译后的 HotSpot 根目录，在根目录中包含 linux_i486_compiler1 和 linux_i486_compiler2。其中 linux_i486_compiler1 目录下的内容是编译后的 Client VM，而 linux_i486_compiler2 目录下的内容则是编译后的 Server VM。

在 jvmg 目录中执行命令“./test_gamma”则可验证 HotSpot 的编译结果，如下所示：

```
root@johngao-virtual-machine:/home/johngao/openjdk7/build-jvm/
  linux_i486_compiler2/jvmg# ./ test_gamma
Using java runtime at: /home/johngao/openjdk7/build/j2sdk-image/jre
openjdk version "1.7.0-internal"
OpenJDK Runtime Environment (build 1.7.0-internal-root_2014_06_28_15_51-b00)
OpenJDK Server VM (build 24.0-b56-internal-jvmg, mixed mode)

 1. A1 B5 C8 D6 E3 F7 G2 H4
 2. A1 B6 C8 D3 E7 F4 G2 H5
 3. A1 B7 C4 D6 E8 F2 G5 H3
 4. A1 B7 C5 D8 E2 F4 G6 H3
 5. A2 B4 C6 D8 E3 F1 G7 H5
 6. A2 B5 C7 D1 E3 F8 G6 H4
88. A7 B5 C3 D1 E6 F8 G2 H4

—省去部分日志内容—

90. A8 B2 C5 D3 E1 F7 G4 H6
91. A8 B3 C1 D6 E2 F5 G7 H4
92. A8 B4 C1 D3 E6 F2 G7 H5
```

尽管已经成功编译好 HotSpot，但是在运行之前，我们还需要修改 jvmg 目录下的 env.sh

脚本。该脚本中已经包含了 JAVA_HOME、CLASSPATH 和 HOTSPOT_BUILD_USER 这 3 个环境变量，但还需新增一个 LD_LIBRARY_PATH 环境变量，如下所示：

代码 1-6　设置 LD_LIBRARY_PATH 环境变量

```
: ${JAVA_HOME:=/home/johngao/openjdk7/build/j2sdk-image}
CLASSPATH=.:${JAVA_HOME}/jre/lib/rt.jar:${JAVA_HOME}/jre/lib/i18n.jar
HOTSPOT_BUILD_USER="root in hotspot"
LD_LIBRARY_PATH=.:${JAVA_HOME}/jre/lib/i386/native_threads:
   ${JAVA_HOME}/jre/lib/i386
export JAVA_HOME CLASSPATH HOTSPOT_BUILD_USER LD_LIBRARY_PATH
```

当成功配置好 LD_LIBRARY_PATH 环境变量后，则可以执行 env.sh 脚本启动 HotSpot，并使用命令“gamma -version”查看由自己亲手编译的 HotSpot 的版本信息，如下所示：

```
root@johngao-virtual-machine:/home/johngao/openjdk7/build-jvm/
  linux_i486_compiler2/jvmg#     . ./env.sh
root@johngao-virtual-machine:/home/johngao/openjdk7/build-jvm/
  linux_i486_compiler2/jvmg# ./ gamma -version
Using java runtime at: /home/johngao/openjdk7/build/j2sdk-image/jre
openjdk version "1.7.0-internal"
OpenJDK Runtime Environment (build 1.7.0-internal-root_2014_06_28_15_51-b00)
OpenJDK Server VM (build 24.0-b56-internal-jvmg, mixed mode)
```

1.5.7　导致编译失败的一些疑难杂症

如果你是第一次尝试编译整个 OpenJDK 或者单独编译 HotSpot，几乎很难做到一次性顺利通过编译。编译失败倒也在情理之中，因为构建编译环境时项目依赖较多（比如：Bootstrap JDK、GCC、Make 和 Ant 等一系列的项目依赖），缺少依赖或者版本问题都会影响编译结果。本书列举了 3 种比较常见的编译错误以及对应的解决方案，供大家参考。

1．项目依赖不完整

在执行编译前，大家尽量确保所有的项目依赖都已经成功下载并安装。如果需要在编译过程中自动下载项目依赖，则可以添加环境变量 ALLOW_DOWNLOADS，如下所示：

```
export ALLOW_DOWNLOADS=true
```

2．项目依赖版本问题

在编译 OpenJDK7 的时候，大家一定要注意 Bootstrap JDK、GCC、Make 和 Ant 等依赖的版本。其中 Bootstrap JDK 的版本必须使用 jdk-update-14 或更高版本，GCC 编译器的版本必须使用 4.3 或更高版本，而 Make 工具的版本必须使用 3.8.1 或更高版本，最后 Ant 工具的版本必须使用 1.7.1 或更高版本。

3．Linux 内核版本过高

我们可以打开/hotspot/make/linux 目录中的 Makefile 脚本，查看 SUPPORTED_OS_VERSION 所支持的所有 Linux 内核版本号。如果你当前的 Linux 内核版本高于 Makefile 脚本中所能支持的上限，则可以在末尾添加实际使用的 Linux 版本号，如下所示：

```
SUPPORTED_OS_VERSION = 2.4% 2.5% 2.6% 3.2%
```

除了可以使用上述方式外，还可以在 Makefile 脚本中，通过注释代码的方式，在编译时绕开 Linux 内核版本的检查步骤，如下所示：

```
check_os_version:
#ifeq ($(DISABLE_HOTSPOT_OS_VERSION_CHECK)$(EMPTY_IF_NOT_SUPPORTED),)
#       $(QUIETLY) >&2 echo "*** This OS is not supported:" `uname -a`; exit 1;
#endif
```

尽管本书列举了 3 种比较常见的编译错误以及对应的解决方案，但编译过程中仍然有太多无法预料的问题，所以当出现了其他问题而无法通过上述解决方案进行解决时，则需要大家寻求其他帮助。

1.5.8 使用GDB工具Debug HotSpot

GDB 是 GUN 开源组织发布的一款用于在 Unix/Linux 下调试 C/C++代码的脚本调试工具，GDB 不仅使用简单且功能强大，只需牢记 GDB 的常用调试命令便可以熟练地使用 GDB 工具。当然除了可以使用 GDB 工具 Debug HotSpot 外，我们还可以使用装有插件的 IDE 工具，比如：Eclipse、NetBeans 等。尽管使用拥有图形化界面的 IDE 工具会更加方便，但使用 GDB 工具却更加灵活，所以本书就以 GDB 为例，为大家演示如何在 Linux 环境下 Debug HotSpot。

既然是 Debug HotSpot，那么调试脚本则是必不可少的。在 jvmg 目录中的 hotspot 就是调试脚本，脚本中设定有大量的调试信息方便大家进行调试，如下所示：

代码 1-7 hotspot 脚本中的调试信息

```
# This is the name of the gdb binary to use
if [ ! "$GDB" ]
then
    GDB=gdb
fi

# This is the name of the gdb binary to use
if [ ! "$DBX" ]
then
    DBX=dbx
```

```
fi
# This is the name of the Valgrind binary to use
if [ ! "$VALGRIND" ]
then
    VALGRIND=valgrind
fi

# This is the name of Emacs for running GUD
EMACS=emacs

# Make sure the paths are fully specified, i.e. they must begin with /.
REL_MYDIR=`dirname $0`
MYDIR=`cd $REL_MYDIR && pwd`

# Look whether the user wants to run inside gdb
case "$1" in
    -gdb)
        MODE=gdb
        shift
        ;;
    -gud)
        MODE=gud
        shift
        ;;
    -dbx)
        MODE=dbx
        shift
        ;;
    -valgrind)
        MODE=valgrind
        shift
        ;;
    *)
        MODE=run
        ;;
esac

# HotSpot 依赖的 JDK 路径
JDK=
if [ "${ALT_JAVA_HOME}" = "" ]; then
    . ${MYDIR}/jdkpath.sh
else
    JDK=${ALT_JAVA_HOME%%/jre};
fi
if [ "${JDK}" = "" ]; then
    echo Failed to find JDK. ALT_JAVA_HOME is not set or ./
      jdkpath.sh is empty or not found.
    exit 1
fi
```

```
# We will set the LD_LIBRARY_PATH as follows:
#     o       $JVMPATH (directory portion only)
#     o       $JRE/lib/$ARCH
# followed by the user's previous effective LD_LIBRARY_PATH, if
# any.
JRE=$JDK/jre
JAVA_HOME=$JDK
export JAVA_HOME

ARCH=i386
SBP=${MYDIR}:${JRE}/lib/${ARCH}

# Set up a suitable LD_LIBRARY_PATH or DYLD_LIBRARY_PATH
OS=`uname -s`
if [ "${OS}" = "Darwin" ]
then
    if [ -z "$DYLD_LIBRARY_PATH" ]
    then
        DYLD_LIBRARY_PATH="$SBP"
    else
        DYLD_LIBRARY_PATH="$SBP:$DYLD_LIBRARY_PATH"
    fi
    export DYLD_LIBRARY_PATH
else
    # not 'Darwin'
    if [ -z "$LD_LIBRARY_PATH" ]
    then
        LD_LIBRARY_PATH="$SBP"
    else
        LD_LIBRARY_PATH="$SBP:$LD_LIBRARY_PATH"
    fi
    export LD_LIBRARY_PATH
fi

JPARMS="$@ $JAVA_ARGS";

# Locate the gamma development launcher
LAUNCHER=${MYDIR}/gamma
if [ ! -x $LAUNCHER ] ; then
    echo Error: Cannot find the gamma development launcher \"$LAUNCHER\"
    exit 1
fi

GDBSRCDIR=$MYDIR
BASEDIR=`cd $MYDIR/../../.. && pwd`

init_gdb() {
# Create a gdb script in case we should run inside gdb
```

```
    GDBSCR=/tmp/hsl.$$
    rm -f $GDBSCR
    cat >>$GDBSCR <<EOF
cd `pwd`
handle SIGUSR1 nostop noprint
handle SIGUSR2 nostop noprint
set args $JPARMS
file $LAUNCHER
directory $GDBSRCDIR

# 当执行到 InitializeJVM 函数的时候执行断点
break InitializeJVM
run
# Stop in InitializeJVM
delete 1
# We can now set breakpoints wherever we like
EOF
}

case "$MODE" in
   gdb)
  init_gdb
       $GDB -x $GDBSCR
  rm -f $GDBSCR
       ;;
   gud)
  init_gdb

# First find out what emacs version we're using, so that we can
# use the new pretty GDB mode if emacs -version >= 22.1
  case `$EMACS -version 2> /dev/null` in
      *GNU\ Emacs\ 2[23]*)
      emacs_gud_cmd="gdba"
      emacs_gud_args="--annotate=3"
      ;;

      *)
      emacs_gud_cmd="gdb"
      emacs_gud_args=
      ;;
  esac
       $EMACS --eval "($emacs_gud_cmd \"$GDB $emacs_gud_args -x $GDBSCR\")";
  rm -f $GDBSCR
       ;;
    dbx)
       $DBX -s $MYDIR/.dbxrc $LAUNCHER $JPARAMS
       ;;
    valgrind)
       echo Warning: Defaulting to 16Mb heap to make Valgrind run faster,
```

```
          use -Xmx for larger
        heap
        echo
        $VALGRIND --tool=memcheck --leak-check=yes --num-callers=
         50 $LAUNCHER -Xmx16m $JPARMS
        ;;
    run)
        LD_PRELOAD=$PRELOADING exec $LAUNCHER $JPARMS
        ;;
    *)
        echo Error: Internal error, unknown launch mode \"$MODE\"
        exit 1
        ;;
esac
RETVAL=$?
exit $RETVAL
```

在 Debug HotSpot 之前，我们还需要检查 hotspot 脚本中所依赖的 JDK 版本。打开 jdkpath.sh 脚本即可查看到依赖的 JDK 版本信息，如下所示：

```
JDK=/home/johngao/openjdk7/build/j2sdk-image
```

当确认好 JDK 的版本后，便可使用命令“./hotspot -gdb TestDemo”进行 Debug HotSpot，如下所示：

```
root@johngao-virtual-machine:/home/johngao/openjdk7/build-jvm/
   linux_i486_compiler2/jvmg# ./ hotspot -gdb TestDemo
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>.
Breakpoint 1 at 0x804bd5d: file /home/johngao/openjdk7/src/openjdk/
  hotspot/src/share/tools/launcher/java.c, line 1270.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
Using java runtime at: /home/johngao/openjdk7/build/j2sdk-image/jre
[New Thread 0xb6a15b40 (LWP 6321)]
[Switching to Thread 0xb6a15b40 (LWP 6321)]

Breakpoint 1, InitializeJVM (pvm=0xb6a152c8, penv=0xb6a152cc, ifn=0xb6a152c0)
  at /home/johngao/openjdk7/src/openjdk/hotspot/src/share/tools/launcher
     /java.c:1270
1270      memset(&args, 0, sizeof(args));
```

```
(gdb) list
1265  InitializeJVM(JavaVM **pvm, JNIEnv **penv, InvocationFunctions *ifn)
1266  {
1267      JavaVMInitArgs args;
1268      jint r;
1269
1270      memset(&args, 0, sizeof(args));
1271      args.version  = JNI_VERSION_1_2;
1272      args.nOptions = numOptions;
1273      args.options  = options;
1274      args.ignoreUnrecognized = JNI_FALSE;
(gdb)
```

由于在 hotspot 脚本中设置了断点操作，所以当 HotSpot 执行到 InitializeJVM()函数时则会停止下来。如果希望 HotSpot 继续执行下一行代码，则可以使用命令“step”或者“next”。其中命令“step”类似于 Eclipse Debug 的按键 F5，也就是说该命令会进入函数。而命令“next”则类似于按键 F6，但是不会进入函数。如下所示：

```
(gdb) next
1271      args.version  = JNI_VERSION_1_2;
(gdb) next 5
1288      r = ifn->CreateJavaVM(pvm, (void **)penv, &args);
(gdb) step
[New Thread 0x9f7d4b40 (LWP 6296)]
[New Thread 0x9f753b40 (LWP 6297)]
[New Thread 0x9f5ffb40 (LWP 6298)]
[New Thread 0x9f5aeb40 (LWP 6299)]
[New Thread 0x9edffb40 (LWP 6300)]
[New Thread 0x9ebffb40 (LWP 6301)]
[New Thread 0x9f55db40 (LWP 6302)]
[New Thread 0x9e9ffb40 (LWP 6303)]
1289      JLI_MemFree(options);
(gdb)
```

如果希望程序直接运行，则可以使用命令“continue”，如下所示：

```
(gdb) continue
Continuing.
Hello HotSpot...
[Thread 0x9e9ffb40 (LWP 6303) exited]
[Thread 0x9f5aeb40 (LWP 6299) exited]
[Thread 0x9f7d4b40 (LWP 6296) exited]
[Thread 0xb6a15b40 (LWP 6295) exited]
[Thread 0x9ebffb40 (LWP 6301) exited]
[Thread 0x9edffb40 (LWP 6300) exited]
[Thread 0x9f5ffb40 (LWP 6298) exited]
[Thread 0x9f753b40 (LWP 6297) exited]
[Thread 0xb6a16700 (LWP 6292) exited]
```

```
[Inferior 1 (process 6292) exited normally]
```

当 HotSpot 的代码全部执行完后，便会输出 Java 对象 TestDemo 中的字符串信息。只要大家熟练掌握 GBD 工具的使用，就可以结合 HotSpot 的源代码尽情 Debug HotSpot，相对于枯燥乏味地直接阅读源代码，采用 Debug 的方式则更容易加深大家对 HotSpot 底层实现原理的理解。

1.6 本章小结

本章笔者详细地介绍了 Java 的体系结构、Java 运行环境的安装和配置，以及 Java 技术未来的发展趋势，让大家对 Java 技术有了一个整体的认识和定位。而在本章的实战小节中，笔者更是重点讲解了如何编译出一套自己的 OpenJDK，并使用 GDB 工具 Debug HotSpot。作为全书的开篇，笔者在本章中并未安插一些较为复杂的概念或技术，而是会在后续章节中循序渐进，为大家讲解字节码的编译原理、HotSpot 的初始化过程、内存分配与垃圾回收、执行引擎等于虚拟机相关的方方面面，让大家更为全面地了解 Java 虚拟机。

第 2 章

字节码的编译原理

Java 最初诞生的目的就是为了在不依赖于特定的物理硬件和操作系统环境下运行，那么也就是说 Java 程序实现跨平台特性的基石其实就是字节码。Java 之所以能够解决程序的安全性问题、跨平台移植性等问题，最主要的原因就是 Java 源代码的编译结果并非是本地机器指令，而是字节码。当 Java 源代码成功编译成字节码后，如果想在不同的平台上面运行，则无需再次编译，也就是说 Java 源码只需一次编译就可处处运行，这就是“Write Once，Run Anywhere”的思想。所以注定了 Java 程序在任何物理硬件和操作系统环境下都能够顺利运行，只要对应的平台装有特定的 Java 运行环境，Java 程序都可以运行，虽然各个平台的 Java 虚拟机内部实现细节不尽相同，但是它们共同执行的字节码内容却是一样的。那么想要让一个 Java 程序正确地运行在 JVM 中，Java 源码就必须要被编译为符合 JVM 规范的字节码。关于规范，开发人员在使用 Java 语言编写一个 Java 程序时需要遵循 Java 语法规范，而将源码编译为字节码的时候又需要符合 JVM 规范，简单来说，前端编译器的主要任务就是负责将符合 Java 语法规范的 Java 代码转换为符合 JVM 规范的字节码文件。

2.1 javac编译器简介

本书在 1.2.2 小节中，对 Java 源码的编译结果做了一个简单介绍，那么从本章开始直至第 3 章我们就一起来深入探讨这个话题。Java 源码的编译结果为什么不是跟 C/C++一样的本地机器指令，而是字节码呢？其实相信大家也都知道，这与 Java 与生俱来的与平台无关性是密不可分的，Java 之所以能够实现“Write Once，Run Anywhere”（一次编译，处处运行），最主要的就是归功于字节码。简单来说，字节码就相当于是一份通用的契约，尽管不同平台上的 Java 虚拟机的实现细节不尽相同，但是它们共同执行的字节码内容却是一样的，这也就是为什么 Java 的设计者们会将 Java 的编译结果设定为具有平台通用性的字节码而非本地机器指令的目的所在。

Java 源码的编译结果屏蔽了与底层操作系统和物理硬件相关的一些特性，使得开发人员尽可能地只需关注于自身业务。因为只有这样，体系结构中立、与平台无关等特性才能够真正地构建起来，否则 Java 技术就是一门传统的静态编译型语言，而非今天的动态编译型语言。或许时至今日仍然有许多非 Java 开发人员在质疑 Java 的程序运行性能，但是请大家相信，Java 早已不再是遭人唾弃，贴上低效性能标签的一门编程语言。其实说这些话的人，大部分都并不了解 Java 技术，本书在 1.1.3 节中曾经提及过，在 JDK1.0 版本时，Java 程序的运行效率确实不尽如人意，这是因为 Java 当时的解释器非常低效，但这却并不代表 Java 语言自身低效，也就是说，程序的运行性能与编程语言其实是没有多大直接关系，真正决定程序运行性能的是编译器。不过在当时，由于技术限制等原因，JVM 中只有解释器，而没有如今先进高效的 JIT 编译器，可是现在，Java 突破了这些技术限制，程序的运行性能已经达到了可以和 C/C++程序一较高下的地步，并且 Java 技术自身的诸多优势同样也是 C/C++无法比拟的，所谓各有所长就是这个道理。在此大家需要注意，世界上永远没有最好的编程语言，只有最适用于具体应用场景的编程语言。

Java 源码的编译结果是字节码，那么肯定需要有一种编译器能够将 Java 源码编译为字节码，承担这个重任的就是配置在“PATH”环境变量中的 javac 编译器。javac 是一种能够将 Java 源码编译为字节码的前端编译器，由于笔者会在本书的第 8 章中剖析 JIT 编译器，因此熟悉 javac 编译器的编译原理则成为了一个必不可缺的铺垫。

2.1.1 javac与Eclipse Compiler for Java编译器

HotSpot VM 并没有强制要求前端编译器只能使用 javac 来编译字节码，其实只要编译结果符合 JVM 规范都可以被 JVM 所识别，所以在 Java 的前端编译器领域，除了 javac 之外，还有一种被大家经常用到的前端编译器，那就是内置在 Eclipse 中的 ECJ（Eclipse Compiler for Java）编译器。相信有不少开发人员会误以为 Eclipse 中同样也是使用 JDK 或者 OpenJDK 中的 javac 编译器来编译字节码，其实不是这样的，Eclipse 中所使用的 ECJ 前端编译器完全是自主研发的，并且可以和 javac 相媲美，甚至可以比 javac 更优秀。

和 javac 的全量式编译不同，ECJ 是一种增量式编译器。在 Eclipse 中，当开发人员编写完代码后，使用“Ctrl+S”快捷键时，ECJ 编译器所采取的编译方案是把未编译部分的源码逐行进行编译，而非每次都全量编译。因此 ECJ 的编译效率会比 javac 更加迅速和高效，当然编译质量和 javac 相比其实大致还是一样的。这里大家需要注意，前端编译器并不会直接涉及编译优化等方面的技术，而是将这些具体优化细节移交给 HotSpot 的 JIT 编译器负责。

ECJ 不仅是 Eclipse 的缺省内置前端编译器，在 Tomcat 中同样也是使用 ECJ 编译器来编译 jsp 文件。由于 ECJ 编译器是采用 GPL v2 的开源协议进行源代码公开，所以，大家可以登录 http://archive.eclipse.org/下载 ECJ 编译器的源码进行阅读和学习甚至是二次开发。

2.1.2 javac的使用与标准选项配置

如果你是第一次使用 javac 编译器来将 Java 的源代码编译为字节码，那么可以在控制台中输入命令“javac -help”查阅 javac 编译器的具体使用方式和一些标准选项配置。

具体用法:

```
javac <选项> <源文件>
```

其中选项包括:

-g	生成所有调试信息
-g:none	不生成任何调试信息
-g:{lines,vars,source}	只生成某些调试信息
-nowarn	不生成任何警告
-verbose	输出有关编译器正在执行的操作的消息
-deprecation	输出使用已过时的 API 的源位置
-classpath <路径>	指定查找用户类文件和注释处理程序的位置
-cp <路径>	指定查找用户类文件和注释处理程序的位置
-sourcepath <路径>	指定查找输入源文件的位置
-bootclasspath <路径>	覆盖引导类文件的位置
-extdirs <目录>	覆盖安装的扩展目录的位置
-endorseddirs <目录>	覆盖签名的标准路径的位置
-proc:{none,only}	控制是否执行注释处理和/或编译
-processor <class1>[,<class2>,<class3>...]	要运行的注释处理程序的名称; 绕过默认的搜索进程
-processorpath <路径>	指定查找注释处理程序的位置
-d <目录>	指定存放生成的类文件的位置
-s <目录>	指定存放生成的源文件的位置
-implicit:{none,class}	指定是否为隐式引用文件生成类文件
-encoding <编码>	指定源文件使用的字符编码
-source <版本>	提供与指定版本的源兼容性
-target <版本>	生成特定 VM 版本的类文件
-version	版本信息
-help	输出标准选项的提要
-Akey[=value]	传递给注释处理程序的选项
-X	输出非标准选项的提要
-J<标志>	直接将 <标志> 传递给运行时系统

本书在 2.1.1 节中介绍了全量式编译和增量式编译的区别，其实大家根本无需担心全量式编译所带来的性能问题，因为在实际的程序开发过程中，我们更多的是在 IDE 工具中使用增量式编译器，并且就算是全量编译过程耗时较久，也并不会影响到程序在运行时的执行性能。

2.1.3 编译原理

本书在 1.4.2 节中曾经提及过 JVM 并不会与 Java 语言“终生绑定”，任何语言编写的程序都可以运行在 JVM 中，前提是源码的编译结果满足并包含 Java 虚拟机的内部指令集、符号表以及其他的辅助信息，它就是一个有效的字节码文件，就能够被虚拟机所识别并装载运行。

当弄清楚这些基本概念之后，接下来我们再来了解 javac 的编译原理，其实所谓编译原理，大家可以理解为 Java 源码编译为字节码时所需要经历的一些编译步骤。javac 编译器在将 Java 源码编译为一个有效的字节码文件，主要会经历 4 个步骤，分别是：词法解析→语法解析→语义解析→生成字节码，如图 2-1 所示。

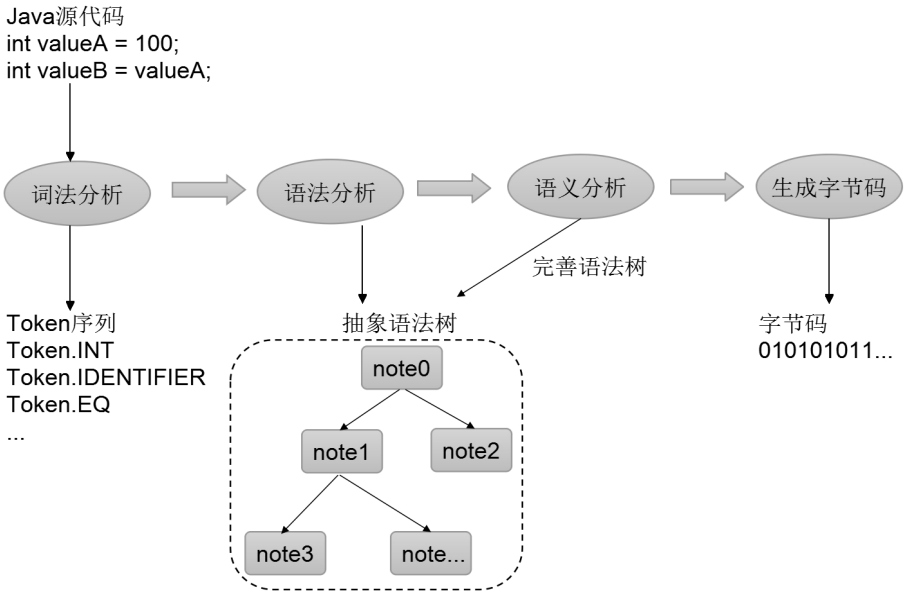


图 2-1 javac 的编译步骤

编译的第一步是词法解析，那么究竟什么是词法解析呢？在 Java 语言中，关键字相信大家都非常熟悉，所谓关键字指的就是 Java API 内部预定义的一些字符集合（如 public、private、class 等）。那么词法解析要做的事情就是将 Java 源码中的关键字和标示符等内容转换为符合 Java 语法规范的 Token 序列，然后按照指定的顺序规则进行匹配校验，这就是词法解析步骤。

当词法解析执行结束后，javac 就会进入到编译的第二个阶段，也就是语法解析。所谓语法解析指的就是将词法解析后的 Token 序列整合为一棵结构化的抽象语法树，因为我们知道，一个 try 语句后面肯定会接上一个 catch 或者 finally 子句，这就是语法解析步骤。

当通过语法解析步骤后，javac 就会进入到生成字节码之前的最后一个阶段，也就是语义解析。由于语法解析器所解析出来的语法树并不能直接应用于生成字节码文件，这是因为这棵语法树相对来说还并不完善，所以语义解析的目的就是为了将之前语法解析步骤所产生的语法树扩充得更加完善，后续编译器将会使用语义解析后的语法树来生成字节码。

当 javac 完成词法解析、语法解析和语义解析等 3 个步骤后，最后一步就是执行字节码编译。这个步骤无疑是整个编译过程中最重要和最核心的地方，不过大家只有在彻底弄清楚前三个编译步骤之后，才能真正理解 javac 编译器的编译原理。

2.1.4 下载javac编译器源码

在正式开始阅读和学习 javac 编译器的源码之前，大家首先需要下载 javac 的源码。javac 的源码包含在 OpenJDK 中，所以如果你并没有下载 OpenJDK 的话，可以先从 http://download.java.net/openjdk/jdk7 中进行下载，或者单独只下载 javac 的源码。

在 OpenJDK 中，javac 的源码包含在/langtools/src/share/classes/com/sun/tools/javac 目录下，如图 2-2 所示。

图 2-2 javac 源码目录结构

对 JVM 有所了解的开发人员都应该知道，HotSpot 其实是使用了 C++以及混合了少量的 C 和汇编代码编写而成的，因此在看到 javac 编译的源码时，或许你会感到有些许诧异，为什么 javac 的源码实现居然全部都是使用 Java 语言编写的呢？而并非使用 C/C++代码？其实早期版本的 javac 源码确实是采用 C/C++代码编写的，只不过在后来除了 HotSpot 中的核心功能模块仍然保持使用 C++以及混合少量的 C 和汇编语言进行编写外，一些比较“外围”的工具早已全部采用 Java 语言进行编写了。对于 Java 开发人员而言，这无疑是一件非常美妙的事情，因为大家再也不必担心因为语法差异等问题所引起的源码阅读障碍。

2.1.5 调用compile()方法执行编译

当你阅读到本节时，笔者完全有理由相信你内心的求知欲望早已蠢蠢欲动，那么接下来，我们就一起深入到 javac 编译器的源码世界中去一探究竟。在此大家需要注意，由于 javac 完全是采用 Java 语言进行编写的，所以考虑到后续的调试跟踪方便，建议大家直接把 javac 的源码整个拷贝到 Eclipse 工具的 Java 工程中。

在正式开始讲解 javac 的编译步骤之前，大家首先找到 com.sun.tools.javac.main.Main 类，该类中提供了一个对外完全透明的字节码编译方法 compile()，这样一来开发人员便可以直

接在程序中调用该方法将目标源文件编译为字节码文件。如下所示：

代码 2-1 调用 compile()方法执行编译

```
/**
* @author JohnGao
*/
public static void main(String[] args) {
  com.sun.tools.javac.main.Main compiler = new com.sun.tools.javac.main.
      Main("javac");
   /* 调用 compile()方法执行编译 */
   compiler.compile(new String[] { "d:/Test01.java" });
}
```

上述代码示例中，compile()方法会将目标源文件中的 Java 代码编译为字节码。其中 compile()的方法参数是一个字符串数组，这也就意味着开发人员不仅可以编译单个 Java 源文件，还可以在数组中添加多个源文件路径，统一调用 compile()方法执行批量编译。如下所示：

代码 2-2 执行批量编译

```
/* 调用 compile()方法执行批量编译 */
compiler.compile(new String[] { "d:/Test01.java", "d:/Test02.java",
   "d:/Test03.java" });
```

不论是编译单个 Java 源文件还是执行批量编译，compile()方法的输出目标都会与源文件一样保持在同级目录下。也就是说，当对源文件进行成功编译后，会在同级目录下生成与源文件名称对应的字节码文件。

在此大家需要注意，compile()方法本身其实并不具备执行编译的能力，而是在运行时由它负责调用其他的编译方法最终完成字节码编译。不过了解 compile()方法则是弄清楚 javac 编译步骤的前提。如下所示：

代码 2-3 compile()方法的完整代码

```
public int compile(String[] args) {
    Context context = new Context();
    javacFileManager.preRegister(context);
      // can't create it until Log has been set up
    int result = compile(args, context);
    if (fileManager instanceof javacFileManager) {
      // A fresh context was created above, so jfm must be a javacFileManager
      ((javacFileManager)fileManager).close();
          }
    return result;
}
```

2.2 词法解析步骤

词法解析是 javac 编译器执行字节码编译的第一步。在词法解析的过程中，词法解析器最主要的任务就是将 Java 源码中的关键字和标示符等内容转换为符合 Java 语法规范的 Token 序列，然后按照指定的顺序规则进行匹配校验，以便为后续的语法解析步骤做准备。

在 javac 编译器中，词法解析器接口是 com.sun.tools.javac.parser.Lexer，它的直接派生实现是位于同包下的 Scanner 类，该类的对象实例由 ScannerFactory 工厂负责创建。Scanner 类的主要任务就是按照单个字符的方式读取 Java 源文件中的关键字和标示符等内容，然后将其转换为符合 Java 语法规范的 Token 序列。在此大家需要注意，负责词法解析工作的并非是 Scanner 类，而是 com.sun.tools.javac.parser.JavacParser 类，该类的对象实例由 ParserFactory 工厂负责创建。也就是说，由 JavacParser 类负责控制词法解析时的具体细节，而 Scanner 类仅仅只是负责读取源码中的字符集合以及与 Token 序列之间的转换任务，如图 2-3 所示。

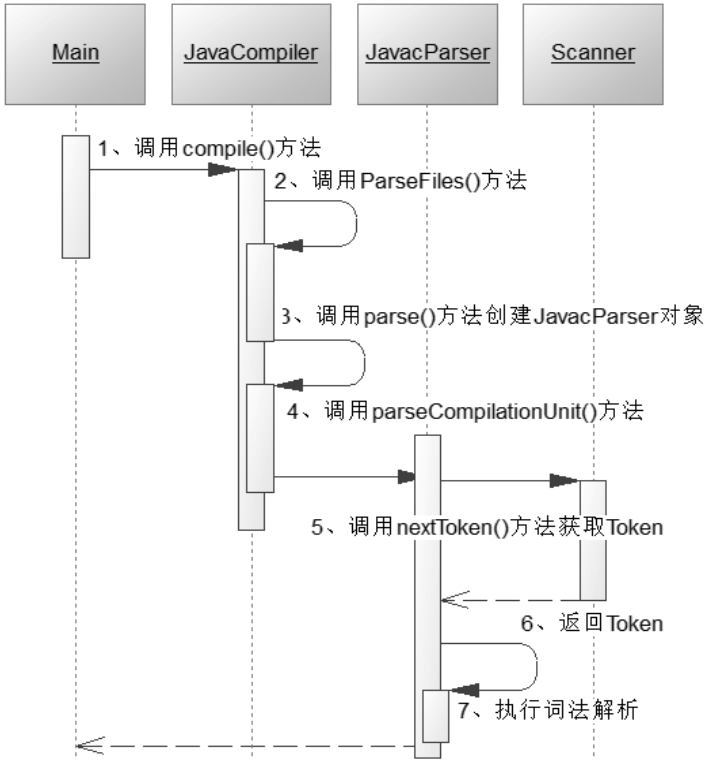


图 2-3 词法解析过程

本书在 2.1.5 节中示例了 com.sun.tools.javac.main.Main 类的 compile()方法的使用，那么接下来我们就开始一步一步地深入到 compile()的方法内部细节中去。

首先由 compile()方法调用 com.sun.tools.javac.main.javacompiler 类的 parseFiles()方法，该方法的主要任务就是调用 parse()方法获取 JavacParser 类的对象实例，然后调用 JavacParser 类的 parseCompilationUnit()方法执行词法解析，如下所示：

代码 2-4 parse()方法的完整代码

```
protected JCCompilationUnit parse(JavaFileObject filename,
      CharSequence content) {
  long msec = now();
  JCCompilationUnit tree = make.TopLevel(
          List.<JCTree.JCAnnotation> nil(), null, List.<JCTree> nil());
  if (content != null) {
      if (verbose) {
          log.printVerbose("parsing.started", filename);
      }
      if (taskListener != null) {
          TaskEvent e = new TaskEvent(TaskEvent.Kind.PARSE, filename);
          taskListener.started(e);
      }
      /* 创建 JavacParser 对象实例 */
      Parser parser = parserFactory.newParser(content, keepComments(),
              genEndPos, lineDebugInfo);
      /* 调用 parseCompilationUnit()方法执行词法解析 */
      tree = parser.parseCompilationUnit();
      if (verbose) {
          log.printVerbose("parsing.done", Long.toString(elapsed(msec)));
      }
  }
  tree.sourcefile = filename;
  if (content != null && taskListener != null) {
      TaskEvent e = new TaskEvent(TaskEvent.Kind.PARSE, tree);
      taskListener.finished(e);
  }
  return tree;
}
```

上述代码示例中，由 ParserFactory 工厂的 newParser()方法负责创建 JavacParser 类的对象实例。JavacParser 类派生于 com.sun.tools.javac.parser.Parser 接口，并重写了 Parser 接口的 parseCompilationUnit()方法。parseCompilationUnit()方法可以说是词法解析步骤的核心所在，由该方法负责调用 Scanner 类的 nextToken()方法获取出 Token 后再按照指定的顺序规则进行匹配校验。

2.2.1 Token序列

词法解析器在成功读取到 Java 源码中的关键字和标示符等内容后，会将其转换为符合 Java 语法规范的 Token 序列，那么究竟什么是 Token 呢？实际上 Token 无非就是一组对应源码字符集合的单词序列。简单来说，Token 其实就一个枚举类型，其内部定义了许多符合 Java 语法规范并与源码字符集合相对应的枚举常量。为了使大家阅读更方便，本书示例了包含在 com.sun.tools.javac.parser.Token 类中的所有枚举常量，如下所示：

代码 2-5 Token 类中的枚举常量

```
EOF,
ERROR,
IDENTIFIER,
ABSTRACT("abstract"),
ASSERT("assert"),
BOOLEAN("boolean"),
BREAK("break"),
BYTE("byte"),
CASE("case"),
CATCH("catch"),
CHAR("char"),
CLASS("class"),
CONST("const"),
CONTINUE("continue"),
DEFAULT("default"),
DO("do"),
DOUBLE("double"),
ELSE("else"),
ENUM("enum"),
EXTENDS("extends"),
FINAL("final"),
FINALLY("finally"),
FLOAT("float"),
FOR("for"),
GOTO("goto"),
IF("if"),
IMPLEMENTS("implements"),
IMPORT("import"),
INSTANCEOF("instanceof"),
INT("int"),
INTERFACE("interface"),
LONG("long"),
NATIVE("native"),
NEW("new"),
PACKAGE("package"),
PRIVATE("private"),
```

```
    PROTECTED("protected"),
    PUBLIC("public"),
    RETURN("return"),
    SHORT("short"),
    STATIC("static"),
    STRICTFP("strictfp"),
    SUPER("super"),
    SWITCH("switch"),
    SYNCHRONIZED("synchronized"),
    THIS("this"),
    THROW("throw"),
    THROWS("throws"),
    TRANSIENT("transient"),
    TRY("try"),
    VOID("void"),
    VOLATILE("volatile"),
    WHILE("while"),
    INTLITERAL,
    LONGLITERAL,
    FLOATLITERAL,
    DOUBLELITERAL,
    CHARLITERAL,
    STRINGLITERAL,
    TRUE("true"),
    FALSE("false"),
    NULL("null"),
    LPAREN("("),
    RPAREN(")"),
    LBRACE("{"),
    RBRACE("}"),
    LBRACKET("["),
    RBRACKET("]"),
    SEMI(";"),
    COMMA(","),
    DOT("."),
    ELLIPSIS("..."),
    EQ("="),
    GT(">"),
    LT("<"),
    BANG("!"),
    TILDE("~"),
    QUES("?"),
    COLON(":"),
    EQEQ("=="),
    LTEQ("<="),
    GTEQ(">="),
    BANGEQ("!="),
    AMPAMP("&&"),
```

```
BARBAR("||"),
PLUSPLUS("++"),
SUBSUB("--"),
PLUS("+"),
SUB("-"),
STAR("*"),
SLASH("/"),
AMP("&"),
BAR("|"),
CARET("^"),
PERCENT("%"),
LTLT("<<"),
GTGT(">>"),
GTGTGT(">>>"),
PLUSEQ("+="),
SUBEQ("-="),
STAREQ("*="),
SLASHEQ("/="),
AMPEQ("&="),
BAREQ("|="),
CARETEQ("^="),
PERCENTEQ("%="),
LTLTEQ("<<="),
GTGTEQ(">>="),
GTGTGTEQ(">>>="),
MONKEYS_AT("@"),
CUSTOM;
```

2.2.2 源码字符集合与Token之间的对应关系

请大家仔细思考一下，由于编译器在执行词法解析过程中，只会对 Token 按照指定的顺序规则进行匹配校验，并不会直接采用源码字符集合的方式执行词法解析，那么在此就产生了两个疑问：首先源码字符集合究竟是如何转换成 Token 的？其次将源码字符集合转换成 Token 后，词法解析器又是采用什么方式保存源码字符集合与 Token 之间的对应关系？

笔者先从第二个疑问开始解答。词法解析器在将源码字符集合转换为 Token 之前，会先将每一个字符集合都转换为一个对应的 Name 对象，也就是说，每一个源码字符集合其实就是一个 Name 对象，然后再由 com.sun.tools.javac.parser.Keywords 类负责实际的 Token 转换任务。Keywords 类会将 Token 中的所有枚举常量全部都转换为 Name 对象，然后将其存储在 Name 对象的内部类 Table 中，这样一来源码字符集合与 Token 之间的对应关系就成功构建起来了，如图 2-4 所示。当成功构建对应关系后，词法解析器还需要一个存储媒介用于保存这些对应关系，那么 Keywords 类中的数组 key 就用于保存源码字符集合与 Token 之间的对应关系。

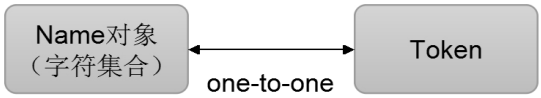


图 2-4 字符集和与 Token 的对应关系

当大家弄清楚源码字符集合与 Token 之间是如何构建起对应关系后，接下来笔者再来为大家解答之前的第一个疑问。对应关系既然是存储在 Keywords 类的数组 key 中的，那么设想一下，既然 Name 对象与 Token 之间构建一种 one-to-one 的对应关系，那么是否通过 Name 对象就可以将源码字符集合转换为对应的 Token？确实是这样的，当词法解析器需要将源码字符集合转换为 Token 时，就会通过 Names 类去调用 Name 类的 fromChars()方法获取一个 Name 对象，然后在调用 Keywords 类的 key()方法时传入这个 Name 对象就可以成功获取出对应的 Token，这样一来词法解析器即可成功地将源码将字符集合转换为 Token。如下所示：

代码 2-6 转换为 Token

```
/* 通过 Names 调用 Name 的 fromChars()方法获取出 Name 对象 */
name = names.fromChars(sbuf, 0, sp);
/* 根据 Name 对象获取出与之对应的 Token */
token = keywords.key(name);
```

2.2.3 调用key()方法获取指定Token

每一个源码字符集合其实就是一个 Name 对象，一旦源码字符集合与 Token 之间成功构建起对应关系后，当词法解析器调用 Keywords 类的 key()方法时，传入与 Token 对应的 Name 对象就可以成功获取指定的 Token。如下所示：

代码 2-7 key()方法的完整代码

```
public Token key(Name name) {
    return (name.getIndex() > maxKey) ? IDENTIFIER : key[name.getIndex()];
}
```

上述代码示例中，源码字符集合与 Token 之间的对应关系就保存在数组 key 中，其中 Name 类的 getIndex()方法用于返回 Name 对象的当前索引，通过这个索引就可以从数组 key 中获取出指定的 Token。

2.2.4 调用nextToken()方法计算Token的获取规则

如果一个源文件中包含 package 关键字声明、import 关键字声明以及 class 主体信息等内容，那么当编译器在执行词法解析步骤时，首先会由词法解析器调用 Keywords 类的 key()

方法将源码字符集合转换为对应的 Token。当成功获取到指定的 Token 后，JavacParser 类就会匹配当前的第一个 Token 是否是 Token.PACKAGE，如果匹配成功的话，再由词法解析器获取出下一个 Token，继续匹配是否是 Token.IDENTIFIER，Token.IDENTIFIER 对应的字符集合就是源码中的标示符，接下来再匹配 Token.DOT、Token.IDENTIFIER 和 Token.SEMI，与这行 Token 序列对应的字符集合就是“package 标示符 1.标示符 2;”。经过这行 Token 序列匹配步骤之后，一个完整的 package 关键字声明就解析完成了，如图 2-5 所示。之后 JavacParser 类还会按照这个匹配规则继续匹配后续的其他 Token 信息。

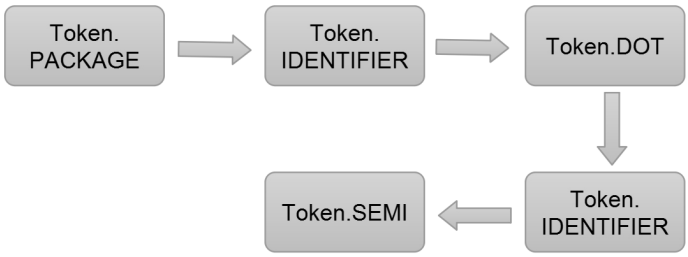


图 2-5 解析 package 关键字声明

不知大家是否发现 Keywords 类的 key()方法仅仅只是根据 Name 对象获取出对应的 Token，然而为什么词法解析器每一次获取的 Token 都符合词法解析的匹配规则？其实调用 key()方法的无非就是 Scanner 类的 nextToken()方法，那么 nextToken()方法中必然会定义符合 Java 语法规范的 Token 读取顺序规则。

2.2.5 调用parseCompilationUnit()方法执行词法解析

词法解析的核心就是校验当前 Token 是否匹配 com.sun.tools.javac.parser.JavacParser 类在 parseCompilationUnit()方法中定义的匹配规则。本书在 2.2.4 节中提及过 Token 的读取顺序是在 Scanner 类的 nextToken()方法中进行控制的，那么当成功读取到 Token 后，parseCompilationUnit()方法便会按照 Java 语法规范校验 Token 的匹配顺序。也就是说，Token 的读取顺序与 Token 的匹配顺序必然需要保持一致，Token 的读取规则决定规定了什么情况下应该出现什么 Token，这样一来，词法解析器便会按照指定的 Token 读取顺序返回与匹配顺序一致的 Token。为了使大家阅读更方便，本书示例了 parseCompilationUnit()方法的完整代码，如下所示：

代码 2-8 parseCompilationUnit()方法的完整代码

```
public JCTree.JCCompilationUnit parseCompilationUnit() {
    int pos = S.pos();
    JCExpression pid = null;
    String dc = S.docComment();
```

```
JCModifiers mods = null;
List<JCAnnotation> packageAnnotations = List.nil();
     /* 解析访问修饰符 */
     if (S.token() == MONKEYS_AT)
    mods = modifiersOpt();

  /* 解析 package 关键字 */
if (S.token() == PACKAGE) {
    if (mods != null) {
        checkNoMods(mods.flags);
        packageAnnotations = mods.annotations;
        mods = null;
    }
    S.nextToken();
    pid = qualident();
    accept(SEMI);
}
ListBuffer<JCTree> defs = new ListBuffer<JCTree>();
boolean checkForImports = true;
while (S.token() != EOF) {
    if (S.pos() <= errorEndPos) {
        // error recovery
        skip(checkForImports, false, false, false);
        if (S.token() == EOF)
            break;
    }
    /* 解析 import 关键字 */
    if (checkForImports && mods == null && S.token() == IMPORT) {
        defs.append(importDeclaration());
    }
    else {
    /* 解析类主体信息并转换为语法树 */
        JCTree def = typeDeclaration(mods);
        if (keepDocComments && dc != null && docComments.get(def) == dc) {
            // If the first type declaration has consumed the first doc
            // comment, then don't use it for the top level comment as well.
            dc = null;
        }
        if (def instanceof JCExpressionStatement)
            def = ((JCExpressionStatement)def).expr;
        defs.append(def);
        if (def instanceof JCClassDecl)
            checkForImports = false;
        mods = null;
    }
}
JCTree.JCCompilationUnit toplevel = F.at(pos).
  TopLevel(packageAnnotations, pid, defs.toList());
```

```
        attach(toplevel, dc);
        if (defs.elems.isEmpty())
            storeEnd(toplevel, S.prevEndPos());
        if (keepDocComments)
            toplevel.docComments = docComments;
        if (keepLineMap)
            toplevel.lineMap = S.getLineMap();
        return toplevel;
    }
```

上述代码示例中，parseCompilationUnit()方法会按照 Token 的匹配规则依次解析出 package 关键字声明、import 关键字声明，当这些 Token 都匹配成功后，词法解析的最后一步就是解析 class 主体信息。直到词法解析步骤全部执行完成后，parseCompilationUnit()方法就会将 Token 序列整合为一棵结构化的抽象语法树。

2.3 语法解析步骤

当词法解析器成功将 Java 源码中的关键字和标示符等内容转换为符合 Java 语法规范的 Token 序列，并按照指定的顺序规则进行匹配校验后，编译器就会进入到下一个编译阶段，即语法解析步骤。所谓语法解析，指的就是将匹配后的 Token 序列整合为一棵结构化的抽象语法树。也就是说，词法解析后的 Token 序列其实还并不完善，因为这些 Token 所代表的只是一个对应的单个源码字符集合，还并没有按照指定的语法规则将其相关的一组或者一段 Token 整合起来。而语法解析的主要任务恰恰就是要将这些零散的 Token 按照指定的 Java 语法规范整合起来形成一个有机的整体，比如一个 try 语句后面肯定会接上一个 catch 或者 finally 子句，这就是语法解析步骤的任务。

在语法解析步骤中，不得不提到的就是 com.sun.tools.javac.tree.JCTree 类。JCTree 类实际上与语法树中的每一个语法节点都保持着密不可分的关系，因为语法树中的每一个语法节点实际上都直接或者间接地继承了 JCTree 类，并且这些语法节点对象都以静态内部类的形式定义在该类中，如图 2-6 所示。

语法树中的每一个语法节点实际上都直接或者间接地继承了 JCTree 类，并且这些语法节点还会实现与节点名称想对应的 Tree 接口，比如语法节点 JCCatch 就实现了 com.sun.source.tree.CatchTree 接口并同时继承了 JCTree 类，JCTree 节点所代表的就是一个异常处理语句中的 cache 子句。

图 2-6 JCTree 与语法节点的继承关系

2.3.1 调用qualident()方法解析package语法节点

当编译器执行词法解析步骤时，parseCompilationUnit()方法会按照 Token 的匹配规则依次解析出 package 关键字声明、import 关键字声明，当这些 Token 都匹配成功后，词法解析的最后一步就是解析 class 主体信息。在此大家需要注意，尽管 parseCompilationUnit()方法会负责控制词法解析时的具体细节，但语法解析步骤其实同样也发生在该方法内部，由它调用其他的语法解析方法执行语法解析。

当词法解析器成功地将 package 关键字声明转换为 Token 并完成词法解析后，就会调用 qualident()方法根据 Token.PACKAGE 解析为 package 语法节点。如下所示：

代码 2-9 qualident()方法的完整代码

```
public JCExpression qualident() {
    /* 解析为 JCIdent 语法节点 */
    JCExpression t = toP(F.at(S.pos()).Ident(ident()));
    while (S.token() == DOT) {
        int pos = S.pos();
        S.nextToken();
        /* 解析为 JCFieldAccess 语法节点 */
        t = toP(F.at(pos).Select(t, ident()));
```

```
        }
        return t;
    }
```

在语法解析步骤中，由 com.sun.tools.javac.tre.TreeMaker 类负责创建 JCTree 类的所有语法节点对象实例，实际上 TreeMaker 类就是一个语法解析器，不过语法解析时的具体细节则是由 parseCompilationUnit()方法进行控制。如果 package 关键字声明中只定义一级目录时，qualident()方法就会调用语法解析器将其解析为一个 JCIdent 语法节点。如下所示：

代码 2-10　Ident()方法的完整代码

```
public JCIdent Ident(Name name) {
        /* 根据 Name 对象解析出一个 JCIdent 语法节点 */
        JCIdent tree = new JCIdent(name, null);
        tree.pos = pos;
        return tree;
}
```

上述代码示例中，由于 package 关键字声明中只定义一级目录，所以语法解析器只会将其解析为一个 JCIdent 语法节点。反之当 package 关键字声明中定义有多级目录时，qualident()方法就会通过循环迭代的方式调用语法解析器将 package 关键字声明解析为嵌套的 JCFieldAccess 语法节点。如下所示：

代码 2-11　Select()方法的完整代码

```
public JCFieldAccess Select(JCExpression selected, Name selector) {
        /* 根据 Name 对象解析出嵌套 JCFieldAccess 语法节点 */
        JCFieldAccess tree = new JCFieldAccess(selected, selector, null);
        tree.pos = pos;
        return tree;
}
```

词法解析器之所以需要将源码字符集合转换为 Token 序列，是因为编译器在执行词法解析过程中只会对 Token 按照指定的顺序规则进行匹配校验，那么语法解析器是否可以直接使用 Token 序列将其解析为语法树或者语法节点呢？仔细观察 Ident()和 Select()方法的方法参数，其实并非是一个 Token，而是与 Token 对应的 Name 对象。也就是说，在解析语法树或者语法节点时，首先需要将 Token 转换为对应的 Name 对象。

在解析 JCIdent 语法节点的 Ident()方法参数中调用了 ident()方法，该方法会返回一个与 Token 对应的 Name 对象。如下所示：

代码 2-12 ident()方法的完整代码

```
Name ident() {
      if (S.token() == IDENTIFIER) {
      Name name = S.name();
      S.nextToken();
      return name;
  } else if (S.token() == ASSERT) {
      if (allowAsserts) {
          error(S.pos(), "assert.as.identifier");
          S.nextToken();
          return names.error;
      } else {
          warning(S.pos(), "assert.as.identifier");
          Name name = S.name();
          S.nextToken();
          return name;
      }
  } else if (S.token() == ENUM) {
      if (allowEnums) {
          error(S.pos(), "enum.as.identifier");
          S.nextToken();
          return names.error;
      } else {
          warning(S.pos(), "enum.as.identifier");
          Name name = S.name();
          S.nextToken();
          return name;
      }
  } else {
      accept(IDENTIFIER);
      return names.error;
  }
}
```

在 qualident()方法中，当调用语法解析器的 Ident()方法解析 JCIdent 语法节点时，其方法参数为 ident()方法的返回值。ident()方法的返回值生成规则是通过 Token.name 的方式将 Token 转换为对应的 Name 对象，这样一来，语法解析器就可以根据 Name 对象成功解析出一个 JCIdent 语法节点。

2.3.2 调用importDeclaration()方法解析import语法树

当成功解析 package 语法节点后，语法解析的下一步就是解析 import 语法树。编译器在执行词法解析过程中，会匹配当前 Token 是否是 Token.IMPORT，如果 Token 匹配成功，parseCompilationUnit()方法就会调用 importDeclaration()方法根据 Token.IMPORT 解析为

import 语法树。如下所示：

代码 2-13 importDeclaration()方法的完整代码

```
JCTree importDeclaration() {
  int pos = S.pos();
  S.nextToken();
  boolean importStatic = false;
  /* 匹配 Token.STATIC */
  if (S.token() == STATIC) {
      checkStaticImports();
      importStatic = true;
      S.nextToken();
  }
  /* 根据 Name 对象解析出一个 JCIdent 语法节点 */
  JCExpression pid = toP(F.at(S.pos()).Ident(ident()));
  do {
      int pos1 = S.pos();
      accept(DOT);
      /* 根据 Name 对象解析出嵌套 JCFieldAccess 语法节点 */
      if (S.token() == STAR) {
          pid = to(F.at(pos1).Select(pid, names.asterisk));
          S.nextToken();
          break;
      } else {
          pid = toP(F.at(pos1).Select(pid, ident()));
      }
  } while (S.token() == DOT);
  accept(SEMI);
  /* 将 JCIdent 和 JCFieldAccess 语法节点整合为一个 JCImport 语法树 */
  return toP(F.at(pos).Import(pid, importStatic));
}
```

上述代码示例中，首先会匹配 Token.STATIC，用于检测 import 关键字声明中是否包含 static 静态导入。接下来 importDeclaration()方法便会调用语法解析器的 Ident()方法解析出一个 JCIdent 语法节点，如果 import 关键字声明中定义多级目录时，则会调用语法解析器的 Select()方法，将其解析为嵌套的 JCFieldAccess 语法节点，这和之前解析 package 语法节点是一样的。

当语法解析器成功解析 JCIdent 和 JCFieldAccess 语法节点后，importDeclaration()方法就会调用语法解析器的 Import()方法，将之前解析过的语法节点整合为一棵 JCImport 语法树，如下所示：

代码 2-14 Import()方法的完整代码

```
public JCImport Import(JCTree qualid, boolean importStatic) {
    /* 解析 JCImport 语法树 */
    JCImport tree = new JCImport(qualid, importStatic);
    tree.pos = pos;
    return tree;
}
```

语法解析器的 Import()方法会将 JCIdent 和 JCFieldAccess 语法节点整合为一棵 JCImport 语法树，这样一来，JCIdent 和 JCFieldAccess 语法节点就会作为 JCImport 语法树的子节点存在。由于在实际的开发过程中，有可能会包含多个 import 关键字声明，那么 parseCompilationUnit()方法内部则会通过循环迭代的方式解析 JCImport 语法树，然后将其存储在一个集合中。

当成功解析 package 和 import 语法节点后，语法解析的最后一步就是将 class 主体信息解析为一棵 JCClassDecl 语法树。

2.3.3 调用classDeclaration()方法解析class语法树

语法解析步骤已经逐渐接近尾声，那么还剩下最后一步就是解析 class 主体信息。当语法解析器成功将 import 关键字声明解析为 JCIdent 和 JCFieldAccess 语法节点并整合为一棵 JCImport 语法树后，在 parseCompilationUnit()的方法内部就会通过 typeDeclaration()方法调用 classOrInterfaceOrEnumDeclaration()方法将 class 主体信息解析为一棵 JCClassDecl 语法树。如下所示：

代码 2-15 classOrInterfaceOrEnumDeclaration()方法的完整代码

```
JCStatement classOrInterfaceOrEnumDeclaration(JCModifiers mods, String dc) {
  /* 匹配 Token.CLASS */
  if (S.token() == CLASS) {
      /* 将类类型解析为一颗 JCClassDecl 语法树 */
      return classDeclaration(mods, dc);
  }
  /* 匹配 Token.INTERFACE */
  else if (S.token() == INTERFACE) {
      /* 将接口类型解析为一颗 JCClassDecl 语法树 */
      return interfaceDeclaration(mods, dc);
  } else if (allowEnums) {
      /* 匹配 Token.ENUM */
      if (S.token() == ENUM) {
          /* 将枚举类型解析为一颗 JCClassDecl 语法树 */
          return enumDeclaration(mods, dc);
```

```
        } else {
            int pos = S.pos();
            List<JCTree> errs;
            if (S.token() == IDENTIFIER) {
                errs = List
                        .<JCTree> of(mods, toP(F.at(pos).Ident(ident())));
                setErrorEndPos(S.pos());
            } else {
                errs = List.<JCTree> of(mods);
            }
            return toP(F.Exec(syntaxError(pos, errs, "expected3", CLASS,
                    INTERFACE, ENUM)));
        }
    } else {
        /* 匹配 Token.ENUM */
        if (S.token() == ENUM) {
            error(S.pos(), "enums.not.supported.in.source", source.name);
            allowEnums = true;
            /* 将枚举类型解析为一颗 JCClassDecl 语法树 */
            return enumDeclaration(mods, dc);
        }
        int pos = S.pos();
        List<JCTree> errs;
        if (S.token() == IDENTIFIER) {
            errs = List.<JCTree> of(mods, toP(F.at(pos).Ident(ident())));
            setErrorEndPos(S.pos());
        } else {
            errs = List.<JCTree> of(mods);
        }
        return toP(F.Exec(syntaxError(pos, errs, "expected2", CLASS,
                INTERFACE)));
    }
}
```

编译器在执行词法解析过程中会校验 classOrInterfaceOrEnumDeclaration()方法中的当前 Token 是否匹配 Token.CLASS，如果匹配成功就调用 classDeclaration()方法将 class 主体信息解析为一棵 JCClassDecl 语法树，否则将继续匹配 Token.INTERFACE 或者 Token.ENUM。

在此大家需要注意，无论当前类型是类类型、接口类型或者是枚举类型，语法解析器都会将其解析为一棵 JCClassDecl 语法树。那么假设 classOrInterfaceOrEnumDeclaration()方法中当前 Token 匹配 Token.CLASS，则会由该方法负责调用 classDeclaration()方法将类类型解析为一颗 JCClassDecl 语法树。如下所示：

代码 2-16 classDeclaration()方法的完整代码

```
JCClassDecl classDeclaration(JCModifiers mods, String dc) {
  int pos = S.pos();
  accept(CLASS);
  Name name = ident();

  List<JCTypeParameter> typarams = typeParametersOpt();

  JCExpression extending = null;
  if (S.token() == EXTENDS) {
      S.nextToken();
      extending = parseType();
  }
  List<JCExpression> implementing = List.nil();
  if (S.token() == IMPLEMENTS) {
      S.nextToken();
      implementing = typeList();
  }
  /* 解析类中的所有成员信息，并存储在集合中 */
  List<JCTree> defs = classOrInterfaceBody(name, false);
  /* 将类中的所有成员信息整合为一棵 JCClassDecl 语法树 */
  JCClassDecl result = toP(F.at(pos).ClassDef(mods, name, typarams,
          extending, implementing, defs));
  attach(result, dc);
  return result;
}
```

上述代码示例中，classBody 的解析由 classOrInterfaceBody()方法负责。那么 classBody 中的内容信息就包括了一个类中的所有变量、方法以及内部类，当成功这些类成员信息全部都解析完成后，就会存储在一个 List 集合中，并由语法解析器的 ClassDef()方法将其整合为一棵 JCClassDecl 语法树。如下所示：

代码 2-17 ClassDef()方法的完整代码

```
public JCClassDecl ClassDef(JCModifiers mods, Name name,
      List<JCTypeParameter> typarams, JCExpression extending,
      List<JCExpression> implementing, List<JCTree> defs) {
  /* 解析 JCClassDecl 语法树 */
  JCClassDecl tree = new JCClassDecl(mods, name, typarams, extending,
          implementing, defs, null);
  tree.pos = pos;
  return tree;
}
```

当成功将 classBody 中的内容信息解析并整合为一棵 JCClassDecl 语法树后，parseCompilationUnit()方法就会调用语法解析器的 TopLevel()方法将之前解析过的 package

语法节点、import 语法树和 class 语法树等内容信息全部整合为一棵 JCCompilationUnit 语法树。如下所示：

代码 2-18　TopLevel()方法的完整代码

```
public JCCompilationUnit TopLevel(List<JCAnnotation> packageAnnotations,
      JCExpression pid, List<JCTree> defs) {
  Assert.checkNonNull(packageAnnotations);
  for (JCTree node : defs)
      Assert.check(
              node instanceof JCClassDecl
                      || node instanceof JCImport
                      || node instanceof JCSkip
                      || node instanceof JCErroneous
                      || (node instanceof JCExpressionStatement
              && ((JCExpressionStatement) node).
              expr instanceof JCErroneous), node.getClass().
              getSimpleName());
  JCCompilationUnit tree = new JCCompilationUnit(packageAnnotations, pid,
          defs, null, null, null, null);
  tree.pos = pos;
  return tree;
}
```

上述代码示例中，JCCompilationUnit 类将会作为整个语法树的根节点，由它持有整个语法树中的所有子节点。如图 2-7 所示。

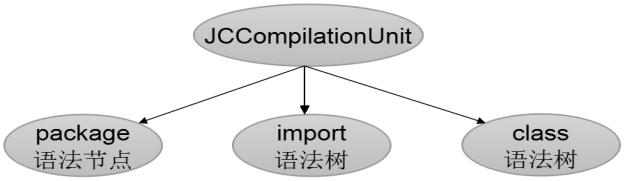


图 2-7　JCCompilationUnit 语法树

当编译步骤经过了词法解析和语法解析后，可以说编译器在生成字节码之前，编译雏形已经成功构建好了。接下来编译器将会进入到生成字节码之前的最后一个阶段，也就是语义解析步骤。

2.4　语义解析步骤

在讲解语义解析步骤之前，我们首先来回顾下词法解析和语法解析这两个编译步骤。词法解析是整个编译步骤的第一个阶段，由词法解析器负责将 Java 源码中的关键字和标示符等内容转换为符合 Java 语法规范的 Token 序列，然后按照指定的顺序规则进行匹配校验。当词法解析结束后，编译器就会进入到编译的第二个阶段，也就是语法解析。所谓语法解析

指的就是将词法解析后的 Token 序列整合为一棵结构化的抽象语法树。在此大家需要注意，由语法解析器解析后的语法树还并不能直接进入到字节码编译阶段，这是因为这颗语法树相对来说还并不完善，而语义解析的任务就是为了将之前语法解析步骤中所产生的语法树扩充得更加完善。在语义解析步骤中会经历的一些操作，如下所示：

- ☐ 为没有构造方法的类型添加缺省的无参构造方法；
- ☐ 检查任何类型的变量在使用前是否都已经经历过初始化；
- ☐ 检查变量类型是否与值匹配；
- ☐ 将 String 类型的常量进行合并处理；
- ☐ 检查代码中的所有操作语句是否可达；
- ☐ 异常检查；
- ☐ 解除 Java 语法糖。

当大家在编写一个 Java 程序时，如果程序中并没有显式地为其定义构造方法，那么编译器在执行语义解析时，会由语义解析器为目标类型添加一个无参的缺省构造方法。如下所示：

代码 2-19 添加缺省构造

```
/**
 * 语义解析之前
 *
 * @author JohnGao
 */
public class Test {
  public static void main(String[] args) {
      new Test();
  }
}

/**
 * 语义解析之后
 *
 * @author JohnGao
 */
public class Test {
  public Test(){}
  public static void main(String[] args) {
      new Test();
  }
}
```

Java 是一门强类型的编程语言，在初始化任何一个变量之前，首先必须明确变量的具体数据类型，所以除了为类型添加无参的缺省构造方法外，语义解析器还会检查指定的变量

类型与所持有的原始值或者引用值之间是否匹配。异常检查同样也属于语义解析步骤时的工作，会由语义解析器负责检查程序中所有的非运行时异常，是否都使用了 try-catch 语句进行捕获，或者是否使用了 throws 关键字向外进行抛出。这里有一个比较有趣的现象，或许这还是大多数开发人员在参加面试时都碰见过的一个问题，那就是一个 String 类型的变量中如果包含多个常量信息并通过符号“+”组合在一起时，底层究竟创建了多少个 String 对象？从表面上来看，相信大家都会误以为 String 的对象数量与常量数量是等价的，可是事实并非如此。编译器在执行语义解析时，会检查 String 变量中是否包含多个常量信息并通过符号“+”组合在一起，如果确实存在，语义解析器则会将其合并为一个字符串，这就是常量折叠操作。如下所示：

代码 2-20　常量折叠

```
/* 语义解析器合并字符串信息之前 */
  String str= "Java 虚拟机精讲," + "作者：高翔龙";
  /* 语义解析器合并字符串信息之后 */
String str= "Java 虚拟机精讲,作者：高翔龙";
```

既然本节谈到了 String 类型，那么笔者再提一个问题，我们都知道 Java 语法层面提供 4 类 8 种原始数据类型，其中使用 Unicode 编码格式的 char 类型可以用于存储世界上所有的文字信息，但该类型的二进制位数只有 16bit，也就是说，char 类型的变量只能够存储单个字符，而 String 类型究竟是如何存储字符串信息的？或许你会想到使用 char 数组的方式来进行存储，其实阅读 String 类的源码后不难发现，String 类型就是 Java 语法层面定义的一种专门用于存储字符串信息的数据结构，其底层实现无非就是替大家实现了这个封装过程而已。

当经历过这一系列的语义解析步骤之后，就构成了一个完善的编译前提，编译器将会使用这个扩充后的语法树将其编译为 Java 字节码。

2.5　生成字节码

成功经历过词法分析、语法分析和语义分析等步骤之后，所解析出来的语法树已经非常完善了，那么 javac 编译器最后的任务就是调用 com.sun.tools.javac.jvm.Gen 类将这棵语法树编译为 Java 字节码文件。其实所谓编译字节码，无非就是将符合 Java 语法规范的 Java 代码转换为符合 JVM 规范的字节码文件。在此大家需要注意，JVM 的架构模型是基于栈的（请阅读第 8 章），也就是说，在 JVM 中所有的操作都需要经过入栈和出栈来完成。

2.6 实战：使用javap工具分析字节码

当 javac 编译器成功将代码 2-19 编译为字节码后，便会在源文件的同级目录下生成与源文件名称对应的后缀名为“.class”的字节码文件。为了更好地理解字节码文件中的内部组成结构，大家可以使用文本编辑器打开字节码文件去一探究竟。本书推荐使用 UltraEdit 编辑器，因为 UltraEdit 不仅使用简单，而且功能强大。

当成功打开字节码文件后，UltraEdit 编辑器便会自动将字节码文件中的内容转换为十六进制数值进行显示。其中前四个字节为 0xCAFEBABE，这 4 个字节所代表的含义就是 magic。简单来说，magic 就是 JVM 用于校验所读取的目标文件是否是一个有效且合法的字节码文件。或许大家会觉得有些奇怪，为什么 JVM 不通过判断文件后缀名的方法来校验字节码文件呢？其实判断文件后缀名也未尝不可，只不过无法确保用户是否会采用手动的方式修改文件的后缀名，所以 Java 的设计者们并没有采用此方案。排列在 magic 后的第 5 个和第 6 个字节所代表的含义就是编译的次版本号，而第 7 个和第 8 个字节就是编译的主版本号。如图 2-8 所示。

图 2-8 字节码文件的组成结构

如果大家细心的话，应该会发现字节码文件中几乎没有使用任何的分隔符区分段落。参考《Java 虚拟机规范（Java SE7 版）》的描述来看，字节码结构组成比较特殊（请阅读第 3 章），其内部并不包含任何的分隔符区分段落，所以无论是字节顺序、数量都是有严格规定的，所有 16 位、32 位、64 位长度的数据都将构造成 2 个、4 个和 8 个 8 位字节单位来表示，多字节数据项总是按照 big-endian 顺序（高位字节在地址最低位，低位字节在地址最高位）来进行存储。也就是说，一组 8 位字节单位的字节流组成了一个完整的字节码文件。

为了更好地分析字节码文件中的内容，大家可以使用 JDK 自带的 javap 工具来反编译字节码文件，在控制台中输入命令“javac -help”即可查阅 javap 工具的具体使用方式和

一些标准选项配置。

具体用法:

```
javap <选项> <classes>
```

其中选项包括:

-help --help -?	打印此用法信息
-version	版本信息
-v -verbose	打印的附加信息
-l	打印行数和局部变量表
-public	仅显示公共类和成员
-protected	显示受保护的公共类和成员
-package	显示包/保护/公共类和成员（默认）
-p -private	显示所有的类和成员
-c	反汇编代码
-s	打印内部类型签名
-sysinfo	显示系统信息（路径，大小，日期，MD5 哈希值）被处理的类
-constants	显示静态常量
-classpath <path>	指定在何处查找用户类文件
-bootclasspath <path>	重写引导类文件的位置

为了验证之前语义解析步骤中语义解析器是否会为没有显式定义构造方法的类型动态添加一个无参的缺省构造方法，以及程序中一个 String 类型的变量如果包含多个字符串信息并通过符号“+”组合在一起时，语义解析器能否会将其合并为一个字符串。那么接下来我们就通过一段简单的 Java 代码来进行验证，如下所示：

代码 2-21　验证语义解析后的变化

```
/**
 * 验证语义解析
 *
 * @author JohnGao
 */
public class DemoTest {
    /**
     * @param agrs
     */
    public static void main(String[] agrs) {
        final String STR = "Java 虚拟机精讲," + "作者：高翔龙";
        System.out.println(STR);
    }
}
```

当成功将上述 Java 源码编译为字节码文件后，在控制台中输入命令“javap -c”即可将字节码文件中的内容进行反编译。如下所示：

代码 2-22 DemoTest 类的字节码指令

```
Compiled from "DemoTest.java"
public class DemoTest {
  public DemoTest();
    Code:
       0: aload_0
       1: invokespecial #1
                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2
                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc #3 // String Java 虚拟机精讲，作者：高翔龙
       5: invokevirtual #4
            // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

在上述反编译后的代码中，语法解析器确实为 DemoTest 类动态添加了一个无参的缺省构造方法，并且 String 变量中的多个通过符号“+”组合在一起的字符串信息也成功合并为了一个字符串。关于字节码文件中的一些字节码指令集，比如 aload_0、invokespecial、getstatic 等，则不在本章的讲解范围之内，如果大家感兴趣，可自行参阅附录 B 中的内容。

2.7 实战：使用GCJ编译器将Java源码直接编译为本地机器指令

尽管大家都知道 Java 源码的编译结果为字节码，但这也并非绝对。因为 Java 源码除了可以使用 ECJ、javac 等前端编译器编译为字节码外，还可以使用 GCJ（GNU Compiler for the Java Programming Language）编译器将其直接编译为本地机器指令运行，也就是说，一旦使用 GCJ 编译器将 Java 源码编译为本地机器指令以后，Java 程序就可以脱离 JVM 环境独立运行。目前有两款比较著名的“元循环”虚拟机（Meta-Circular JVM），分别是 JavaInJava 和 Maxine VM。所谓“元循环”指的就是使用由 Java 语言自身编写的虚拟机去运行 Java 程序，这听起来有些不切实际，因为似乎追溯到了究竟是鸡生蛋还是蛋生鸡的问题，但是这两款虚拟机确确实实地做到了这一点。

JavaInJava 是由 Sun 公司在 1997 年研发的一款带有实验性质的虚拟机。首先这款虚拟

机并非是完全采用 Java 语言编写的，其次 JavaInJava 还需要寄生在宿主虚拟机中才能够运行字节码，也就是说，JavaInJava 其实也只是一个普通的 Java 程序，需要由前端编译器将其编译为字节码文件后，再由宿主虚拟机装载进内部它才能够运行 Java 程序。尽管 JavaInJava 可以运行 Java 程序，但是这款虚拟机内部却并没有内置 JIT 编译器，Java 程序只能依靠解释器逐行解释运行，因此运行效率非常低下。时至今日，JavaInJava 项目几乎已经废弃了。

除了 JavaInJava 之外，还有一款比较著名的“元循环”虚拟机，那就是 Maxine VM。相对于 JavaInJava 来说，Maxine VM 不仅内置有 JIT 编译器，并且仅有 Launcher 是采用 C 语言实现的外，其他核心模块完全是采用 Java 语言自身实现的。最重要的是 Maxine VM 的编译结果为本地机器指令，因此 Maxine VM 完全可以在不依赖于宿主虚拟机的环境下独立运行 Java 程序。

使用 GCJ 编译器将 Java 源码编译为本地机器指令和使用 Javac 编译器编译字节码一样简单。所谓工欲善其事，必先利其器，大家可以登录 http://www.thisiscool.com/gcc_mingw.htm 下载 GCJ 编译器的最新版本，本书所使用的 GCJ 编译器为 4.3。在此需要提醒大家，低版本的 GCJ 编译器不仅不成熟，而且还无法编译中文字符，所以建议大家直接下载和使用高版本的 GCJ 编译器。当下载好 GCJ 编译器并成功将其解压后，首先需要将\thisiscool-gcc\gcc-ecj\bin 目录添加到 PATH 环境变量中，然后打开控制台输入命令“gcj --help”即可查阅 gcj 编译器的具体使用方式和一些标准选项配置。

当成功配置好 GCJ 编译器的环境变量后，大家就跟着笔者一起来尝试着使用 GCJ 编译器将 Java 源码编译为本地机器指令。不过在编译之前，我们首先需要编写一个简单的 Java 程序示例，如下所示：

代码 2-22　一个简单的 Java 程序

```
/**
 * 将 Java 源码直接编译为本地机器指令
 *
 * @author JohnGao
 */
public class GCJTest {
  public static void main(String[] agrs) {
      System.out.println("Hello GCJ 编译器!");
  }
}
```

接下来要做的事情就是准备一个编译脚本，并存放在与 GCJTest.java 文件的同级目录下，编译脚本中的内容就是 gcj 的编译命令。如下所示：

代码 2-23 gcj 的编译命令

```
gcj -c -O3 GCJTest.java
gcj GCJTest.o --main=GCJTest -o GCJTest.exe -O3
```

上述编译命令中，GCJ 编译器首先会根据 Java 源码编译出一个后缀名为“.o”的对象文件，该文件中所包含的内容就是可以直接被 CPU 所识别并执行的本地机器指令。然后 GCJ 编译器再根据这个对象文件 link 出一个后缀名为“.exe”的可执行程序。如图 2-9 所示。

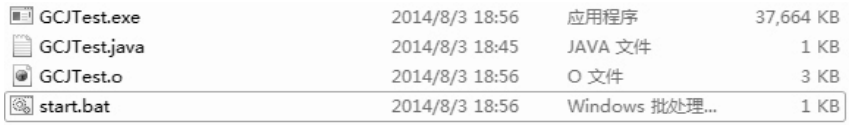


图 2-9 将 Java 源码编译为本地机器指令

除了可以使用 Java 源码编译为本地机器指令外，字节码文件和 jar 构件同样也可以使用 GCJ 编译器进行编译。当成功将 GCJTest.java 编译为可执行程序后，通过鼠标双击 GCJTest.exe 便会执行输出，如图 2-10 所示。

图 2-10 执行编译后的可执行程序

尽管使用 GCJ 编译器可以将 Java 源码编译为本地机器指令，但是在实际的开发过程中笔者却并不建议这么做。因为这样一来 Java 程序将无法实现“Write Once，Run Anywhere”，并且编译为可执行程序之后的 Java 程序体积将会高达几十兆，最关键的是 GCJ 编译器默认只会对 Java 的基础类库提供支持，而其他的第三方构件 GCJ 编译器将无能为力。所以最好还是由 HotSpot 中的 JIT 编译器去负责本地机器指令的动态编译工作。

2.8 本章小结

本章笔者深入源码剖析了 javac 编译器的编译原理，涵盖了编译过程中的词法解析步骤、语法解析步骤、语义解析步骤和生成字节码 4 个主要编译步骤。在词法解析步骤中，本书重点讲解了源码字符集合与 Token 序列之间的转换过程，以及 Token 的获取规则和匹配规则。而在语法解析和语义解析步骤中，本章还讲解了语法解析器如何将 Token 解析为对应的 package 语法节点、import 语法树和 class 语法树，以及语义解析器对语法树的完善扩充，从而最终生成有效的字节码文件。在本章的实战小节中，笔者示例了如何使用 javap 工具将字节码文件反编译后进行分析，以及使用 GCJ 编译器将 Java 源码直接编译为本地机器指令运行。

第 3 章

字节码文件

透过上一章，大家可以明确地知道 Java 代码究竟是如何被前端编译器（javac）成功编译为一个符合 Java 虚拟机规范的字节码文件。字节码结构组成比较特殊，其内部并不包含任何的分隔符区分段落，所以无论是字节顺序、数量都是有严格规定的，所有 16 位、32 位、64 位长度的数据都将构造成 2 个、4 个和 8 个 8 位字节单位来表示，多字节数据项总是按照 big-endian 顺序（高位字节在地址最低位，低位字节在地址最高位）来进行存储的，也就是说，一组 8 位字节单位的字节流组成了一个完整的字节码文件。那么本章的内容将会主要围绕编译后的字节码文件来进行讲解，让大家深刻理解字节码文件的内部组成结构。

3.1 字节码文件的内部组成结构

参考《Java 虚拟机规范 Java SE7 版》的描述来看，每一个字节码文件其实都对应着全局唯一的一个类或者接口的定义信息。字节码文件采用的是一种类似于 C 语言结构体的伪结构来描述字节码文件格式。为了避免与类的字段、实例等概念产生混淆，本书将用于描述类结构格式的内容定义为项（item）。

每一项都包括类型、名称以及该项的数量。类型可以是表名，同时也是“基本类型”。包含在字节码文件中，各项按照严格的顺序进行连续存放，其内部并不包含任何的分隔符区分段落。在此大家需要注意，在这个结构体中只有两种数据结构，分别是无符号数和表，其中无符号数属于字节码文件中的“基本类型”，如表 3-1 所示。

表 3-1 字节码文件中的“基本类型”

u1	1 个字节，无符号类型
u2	2 个字节，无符号类型
u4	4 个字节，无符号类型
u8	8 个字节，无符号类型

表是由多个无符号数或者其他表作为数据项构成的复合数据类型，所有表的后缀都是使用“_info”进行结尾，并且字节码文件实质上也就是一张表。在此大家需要注意，尽管采用的是类似于 C 语言的数组语法来表示表中的项，但是无法直接将字节偏移量作为索引对表进行访问，因为表中每一项的长度都是不固定且可变的。而当描述一个数据结构为数组时，就意味着它由 0 至多个长度固定的项组成，这个时候便可以采用数组索引的方式对其进行访问。每一个字节码文件对应着一个 ClassFile 的结构，如下所示：

代码 3-1 ClassFile 结构

```
ClassFile {
     u4              magic;
     u2              minor_version;
    u2               major_version;
    u2               constant_pool_count;
    cp_info          constant_pool[constant_pool_count-1];
    u2               access_flags;
    u2               this_class;
    u2               super_class;
    u2               interfaces_count;
    u2               interfaces[interfaces_count];
    u2               fields_count;
    field_info       fields[fields_count];
    u2               methods_count;
    method_info      methods[methods_count];
    u2               attributes_count;
    attribute_info   attributes[attributes_count];
}
```

关于 ClassFile 结构的描述信息，如下所示。

（1）magic（魔术）

一个有效的字节码文件的前 4 个字节为 0xCAFEBABE，也被称之为魔术（magic number）。魔术就是 JVM 用于校验所读取的目标文件是否是一个有效且合法的字节码文件。或许大家会觉得有些奇怪，为什么 JVM 不通过判断文件后缀名的方法来校验字节码文件呢？其实判断文件后缀名也未尝不可，只不过无法确保用户是否会采用手动的方式修改文件的后缀名，所以 Java 的设计者们并没有采用此方案。早在 1991 年 Oak 还未被更名为 Java 时，这个 magic 就已经确定下来了，其实选择 0xCAFEBABE 作为字节码文件的 magic 还有一段小插曲，当时 JVM 的设计者们希望以一种好玩的、有趣的，并且容易记忆的东西来代

表 magic，因此选用 0xCAFEBABE 是因为它象征着著名的咖啡品牌 Peet's Coffee 中深受欢迎的 Baristas（一种咖啡的名称），并且它还预示了 Java 这个商标名字的出现。

（2）minor_version（次版本号）和 major_version（主版本号）

紧跟在 magic 之后的 4 个字节就是编译的次版本号和主版本号，它们共同构成了字节码文件的版本号。比如字节码文件的主版本号为 M，次版本号为 m，那么这个字节码文件的版本号就被确定为 M.m，字节码文件版本号大小的顺序为：1.5<2.0<2.1。

随着 Java 技术的不停更新，不断扩充新晋特性，字节码文件的格式必然会随之发生变化，同样的字节码文件中的版本号也会随之发生变化。对于 JVM 而言，字节码文件中的版本号确定了特定的字节码文件格式，通常只有给定主版本号和一系列次版本号之后，JVM 才能够读取一个字节码文件。如果字节码文件的版本号超出了 JVM 所能够处理的有效范围，那么 Java 虚拟机将不会处理这个字节码文件，比如由高版本的 JDK 编译的字节码文件，自然不能在低版本的 JVM 中运行，否则将会抛出 java.lang.UnsupportedClassVersionError 异常，不过高版本的 JVM 却能向下兼容运行由低版本 JDK 编译的字节码文件。

（3）constant_pool_count（常量池计数器）和 constant_pool（常量池）

在字节码文件中，紧跟在次版本号和主版本号后面的就是常量池计数器和常量池。常量池是字节码文件中非常重要的数据项，同时也是字节码文件中与其他项关联最多和占用字节码空间最大的数据项。常量池中主要用于存放字面量（Literal）和符号引用（Symbolic References）两大类数据常量，其访问方式是通过索引来进行访问的，但由于常量池列表中的常量数并不固定，因此在常量池之前就需要通过一个 2 个字节的常量池计数器来统计常量池列表中到底拥有多少常量项。在此大家需要注意，常量池计数器中的计数值并非是从 0 开始进行计数的，而从 1 开始，也就是说，如果常量池中有 2 个常量时，计数值则为 2，依此类推。尽管常量池计数器中并没有将 0 作为计数值，但 0 却仍然存在，这么做的目的就是为了满足后续其他项在不引用常量池中的任何常量项时，默认可以把常量池中的访问索引设置为 0 来表示。

常量池中存放的字面量由文字字符串、final 常量值等构成，而符号引用则包括了类和接口的全限定名（Fully Qualified Name）、字段的名称和描述符（Descriptor），以及方法的名称和描述符。关于字节码文件中的常量池信息，如下所示。

代码 3-2　使用命令“javap -verbose”查看字节码文件中的常量池信息

```
Compiled from "ConstantTest.java"
public class ConstantTest extends java.lang.Object
  SourceFile: "ConstantTest.java"
```

```
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method       #7.#18; //  java/lang/Object."<init>":()V
const #2 = String       #19;    //  Java虚拟机精讲
const #3 = Field        #6.#20; //  ConstantTest.value1:Ljava/lang/String;
const #4 = String       #21;    //  作者：高翔龙
const #5 = Field        #6.#22; //  ConstantTest.value2:Ljava/lang/String;
const #6 = class        #23;    //  ConstantTest
const #7 = class        #24;    //  java/lang/Object
const #8 = Asciz        value1;
const #9 = Asciz        Ljava/lang/String;;
const #10 = Asciz       ConstantValue;
const #11 = Asciz       value2;
const #12 = Asciz       <init>;
const #13 = Asciz       ()V;
const #14 = Asciz       Code;
const #15 = Asciz       LineNumberTable;
const #16 = Asciz       SourceFile;
const #17 = Asciz       ConstantTest.java;
const #18 = NameAndType #12:#13;//  "<init>":()V
const #19 = Asciz       Java虚拟机精讲;
const #20 = NameAndType #8:#9;//  value1:Ljava/lang/String;
const #21 = Asciz       作者：高翔龙;
const #22 = NameAndType #11:#9;//  value2:Ljava/lang/String;
const #23 = Asciz       ConstantTest;
const #24 = Asciz       java/lang/Object;
```

在 C 语言中，如果一个程序需要调用其他库中的函数，那么在链接时，该函数在库中的位置（即相对于库文件开头的偏移量）会被写在程序中，在运行时直接去这个地址调用函数即可。但在 Java 中却并不是这样的，因为字节码文件中并没有包含各个方法和字段的最终内存布局信息，也就是说，只有当 Java 虚拟机在运行时，从常量池中获取出对应的符号引用，并经过解析阶段将这些符号引用全部转换为直接引用后，Java 虚拟机才能正常使用。

之前笔者在 3.1 小节中曾经提及过结构体中的两种数据结构：无符号数和表，表是由多个无符号数或者其他表作为数据项构成的一种复合数据类型，所有表的后缀都是使用“_info”进行结尾的。那么在常量池列表中的每一个常量项其实都是一个表，Java7 一共包含 14 种类型不尽相同的常量项（关于常量项，请阅读 3.3 节），如表 3-2 所示。

表 3-2 常量池中的表类型

类型	tag	描述
CONSTANT_Utf8_info	1	UTF-8 编码的字符串
CONSTANT_Integer_info	3	整型字面值
CONSTANT_Float_info	4	单精度浮点类型字面值
CONSTANT_Long_info	5	长整型字面值
CONSTANT_Double_info	6	双精度浮点类型字面值
CONSTANT_Class_info	7	类或者接口的符号引用
CONSTANT_String_info	8	字符串类型的字面值
CONSTANT_Fieldref_info	9	字段的符号引用
CONSTANT_Methodref_info	10	方法的符号引用
CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或方法的部分符号引用
CONSTANT_MethodHandle_info	15	表示方法句柄
CONSTANT_MethodType_info	16	标示方法类型
CONSTANT_InvokeDynamic_info	18	标示一个动态方法调用点

（4）access_flags（访问标志）

紧跟在常量池之后的 2 个字节就是访问标志，访问标志主要就是用于表示某个类或者接口的访问权限。比如：访问标志指明的是字节码文件中的类还是接口；使用的访问修饰符是哪一种；是否是由 abstract 关键字修饰的抽象类；如果是被 abstract 修饰的抽象类，不能再标记为 final 类型；接口同样也不允许被标记为 final 类型等。访问标志的定义，如表 3-3 所示。

表 3-3 访问标志

标志名称	值	描述
ACC_PUBLIC	0x0001	声明为 public，可以被包的类进行外访问
ACC_FINAL	0x0010	声明为 final，不允许有派生类
ACC_SUPER	0x0020	当用到 invokespecial 指令时，需要特殊处理的超类方法
ACC_INTERFACE	0x0200	标志定义的是接口而不是类
ACC_ABSTRACT	0x0400	声明为 abstract，不能够被实例化
ACC_SYNTHETIC	0x1000	声明为 synthetic，标识并非 Java 代码生成的
ACC_ANNOTATION	0x2000	标识注解类型
ACC_ENUM	0x4000	标识枚举类型

访问标志中总共有 16 个标志可用，不过目前只定义了表 3-3 中的 8 个，对于那些没有被用到的标志，编译器一律要求设置为 0，并且 Java 虚拟机还必须忽略掉它。

（5）this_class（类索引）和 super_class（超类索引）

紧跟在访问标志之后的 4 个字节就是类索引和超类索引，类索引和超类索引各自会通过索引指向常量池列表中一个类型为 CONSTANT_Class_info 的常量项。CONSTANT_Class_info 由 tag 和 name_index 两部分构成，tag 是一个具有 CONSTANT_Class_info 值的常量，而 name_index 则是一个指向常量池列表中类型为 CONSTANT_Utf8_info 常量项的索引，通过这个索引值即可成功获取到 CONSTANT_Utf8_info 常量项中的全限定名字符串，如图 3-1 所示。简单来说，类索引用于确定当前类的全限定名，而超类索引则用于确定当前类的超类全限定名。

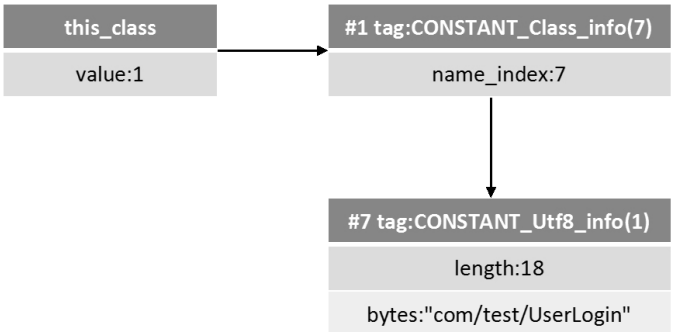


图 3-1 类索引从常量池列表中查找全限定名

在此大家需要注意，Java 语法规范规定了一个类不允许同时继承多个超类，也就是说，超类索引只能有一个。由于 java.lang.Object 是所有类型的超类，那么如果一个类缺省并没有显式地继承任何超类，那么它的直接超类就应该是 Object，其父类索引的值就为 0，这就是之前笔者说过的，为了满足后续其他项在不引用常量池中的任何常量项时，缺省可以把常量池中的访问索引设置为 0 来进行表示。

（6）interfaces_count（接口计数器）和 interfaces（接口表）

紧跟在类索引和超类索引之后的 4 个字节就是接口计数器和接口表。接口计数器用于表示当前类或者接口的直接超类接口数量，在此大家需要注意，在字节码文件中，除了常量池计数器中的计数值从 1 开始计数外，其余的数组集合类型都是从 0 开始计数。而接口表实际上则是一个数组集合，它包含了当前类或者接口在常量池列表中直接超类接口的索引集合，通过这个索引即可确定当前类或者接口的超类接口的全限定名。

（7）fields_count（字段计数器）和 fields（字段表）

紧跟在接口计数器和接口表之后的就是字段计数器和字段表。字段计数器用于表示一个字节码文件中的 field_info 表总数，也就是一个类中类变量和实例变量的数量总和。而字段

表实际上则是一个数组集合，字段表中的每一个成员都必须是一个 field_info 结构的数据项。简单来说，field_info 用于表示一个字段的完整信息，比如字段的标示符、访问修饰符（public、private 或 protected）、是类变量还是实例变量（static 修饰符）、是否是常量（final 修饰符）等。在此大家需要注意，字段表中所包含的字段信息仅限于当前类或接口的所属字段，并不包含继承超类后的字段信息。关于 field_info 项的表结构信息，请阅读 3.4 节。

（8）methods_count（方法计数器）和 methods（方法表）

紧跟在字段计数器和字段表之后的就是方法计数器和方法表。方法计数器用于表示一个字节码文件中的 method_info 表总数。而方法表实际上则是一个数组集合，方法表中的每一个成员都必须是一个 method_info 结构的数据项。简单来说，method_info 用于表示当前类或者接口中某个方法的完整描述，比如方法标示符、方法的访问修饰符（public、private 或 protected）、方法的返回值类型及方法的参数信息等。在此大家需要注意，方法表中所包含的方法信息仅限于当前类或者接口中的所属方法，并不包含继承超类后的方法信息。关于 method_info 项的表结构信息，请阅读 3.5 节。

（9）attributes_count（属性计数器）和 attributes（属性表）

紧跟在方法计数器和方法表之后的就是属性计数器和属性表。属性计数器用于表示当前字节码文件中的 attribute_info 表总数。而属性表同之前的字段表和方法表一样都是一个数组集合，属性表中的每一个成员都必须是一个 attribute_info 结构的数据项。每一个 attribute_info 表的第一项都是指向常量池列表中 CONSTANT_Utf8_info 项的索引，该表给出了属性的名称。

属性可以出现在 ClassFile 表、字段表和方法表中，用以描述与其相关的信息，比如描述字节码文件中所定义的类和接口相关的信息、描述与字段相关的信息、描述与方法相关的信息。关于 attribute_info 项的表结构信息，请阅读 3.6 节。

3.2 符号引用

常量池中主要用于存放字面量（Literal）和符号引用（Symbolic References）两大类数据常量，其访问方式是通过索引来进行访问的，那么符号引用则由 3 种特殊的字符串构成，分别是：全限定名、简单名称和描述符。

在字节码文件中，使用全限定名可以用于描述类或者接口。而类或者接口中定义的字段则都会包含一个简单名称和字段描述符。除此之外，定义在类或者接口中的方法，仍然会包含一个简单名称和方法描述符。

3.2.1 类或者接口的全限定名

字节码文件中包含的所有类或者接口的名称，都是通过全限定名（Fully Qualified Name）的方式来进行表示的，如表 3-1 所示。CONSTANT_Class_info 表中的 name_index 是一个指向常量池列表中一个类型为 CONSTANT_Utf8_info 常量项的索引，通过这个索引值即可成功获取 CONSTANT_Utf8_info 常量项中的全限定名字符串，那么由此可见，类或者接口的名称则被保存在一个类型为 CONSTANT_Utf8_info 常量项中。

参考《Java 虚拟机规范 Java SE7 版》的描述来看，比如一个 Object 类型的全限定名为 java.lang.Object，但是在字节码文件中，全限定名中的符号“.”被符号“/”进行了取代，也就是说，java/lang/Object 才是 Object 类型在字节码文件中的内部表示形式。那么像 Java API 中的一些其他类型，比如接口 Map 在字节码文件中的全限定名则是 java/util/Map，String 类型在字节码文件中的全限定名则是 java/lang/String。

3.2.2 简单名称

在字节码文件中，类或者接口中所定义的所有字段名称和方法名称都是使用简单名称来进行存储的。比如一个指向 java/lang/String 类型中返回类型为 String 的 toString()方法的简单名称就是“toString”。比如一个指向接口 java/util/map 中返回类型为 int 的 size()方法的简单名称就是“size”。在此大家需要注意，字段或者方法的简单名称中不能够包含“.”、“;”、“[”和“/”等以 ASCII 和 Unicode 字符的表示形式。

3.2.3 描述符

类或者接口中定义的字段和方法除了都会包含一个简单名称外，还会包含一个描述符（Descriptor），描述字段的描述符称为字段描述符，描述方法的则称之为方法描述符。简单来说，字段描述符用于描述字段类型，而方法描述符则用于描述方法的返回值类型以及方法的参数数量、类型和参数顺序。字段描述符（Field Descriptor）序列，如下所示：

代码 3-3 字段描述符序列

```
FieldDescriptor:
    FieldType
ComponentType:
    FieldType
FieldType:
    BaseType
    ObjectType
    ArrayType
```

```
BaseType:
    B
    C
    D
    F
    I
    J
    S
    Z
ObjectType:
    L Classname ;
ArrayType:
    [ComponentType
```

上述代码示例中，BaseType 类型中的描述符所代表的含义就是 4 类 8 种基本数据类型，ObjectType 类型中的描述符“L”所代表的含义就是对象类型，而 ArrayType 类型中的描述符“[”所代表的含义就是数组类型。在此大家需要注意，BaseType、ObjectType 以及 ArrayType 类型中的任何一个描述符都是使用 ASCII 字符的表示形式。字段描述符的解释信息，如表 3-4 所示。

表 3-4　字段描述符解释表

描述符	类型	含义
B	byte	字节类型
C	char	字符类型
D	double	双精度浮点类型
F	float	单精度浮点类型
I	int	整型
J	long	长整型
S	short	短整型
Z	boolean	布尔类型
L	reference	对象类型
[	reference	数组类型

通过表 3-4 我们可以明确代码 3-3 中的字段描述符序列，比如字段描述符是“B”的话，那么肯定是用于描述一个 byte 类型的基本数据。如果字段描述符是“Ljava/lang/String;”的话，那么肯定是用于表示一个 java.lang.String 类型的实例（在 ObjectType 中的 Classname 用于表示一个类或者接口的全限定名）。关于数组类型，字段描述符“[C”所代表的含义就是一个 char 类型的一维数组，那么如果是一个 char 类型的二维数组则使用描述符“[[C”进

行表示，依此类推。

当大家理解字段描述符之后，接下来我们再来看看方法描述符。如下所示：

代码 3-4 方法描述符序列

```
MethodDescriptor:
    ( ParameterDescriptor* ) ReturnDescriptor
ParameterDescriptor:
    FieldType
ReturnDescriptor:
    FieldType
    VoidDescriptor
    VoidDescriptor:
        V
```

上述代码示例中，方法描述符（Method Descriptor）中包含了参数描述符（Parameter Descriptor）和返回值描述符（Return Descriptor）。参数描述符用于描述一个方法的参数数量、类型和参数顺序，而返回值描述符则用于描述当前方法的返回值信息，返回值描述符中的 VoidDescriptor 用于表示当前方法并没有任何的返回值类型，仅仅只是一个 void 类型，即描述符“V”。如果被定义的方法为 java.lang.Object getInfo(java.lang.String userName,int age,boolean sex)，那么其对应的方法描述符就应该是“(Ljava/lang/ObjectIZ)Ljava/lang/Object”。如果被定义的方法并没有包含任何的方法参数 java.lang.Object getInfo()，那么其对应的方法描述就应该是“()Ljava/lang/Object”。

参考《Java 虚拟机规范 Java SE7 版》的描述来看，如果一个方法描述符是有效的，那么它对应的方法参数列表的总长度应该小于等于 225，对于实例方法和接口方法，需要额外考虑隐式参数 this，不过这一点并非由方法描述符来进行表示，而是由 Java 虚拟机在调用实例方法所使用的指令中实现 this 参数的隐式传递。

3.3 常量池

笔者之前曾经提及过，常量池列表中的常量数并不固定，因此在常量池之前就需要通过一个 2 个字节的常量池计数器来统计常量池列表中到底拥有多少常量项。Java7 中一共包含 14 种类型不尽相同的常量项，但是每个常量项之间的格式都具有一定的通用性，比如每一个常量项中都包含一个 1 个字节的“tag”项作为开头，它表明了常量项的具体类型和格式，如表 3-2 所示。而后面 info[]项的内容则取决于 tag 的类型。常量池列表中常量项的通用格式，如下所示：

代码 3-5 常量池的通用格式

```
cp_info
{
    u1 tag;
    u1 info[];
}
```

3.3.1 CONSTANT_Utf8_info常量项

在常量池列表中，CONSTANT_Utf8_info 常量项是一种使用改进过的 UTF-8 编码格式来存储诸如文字字符串、类或者接口的全限定名、字段或者方法的简单名称以及描述符等常量字符串信息。CONSTANT_Utf8_info 常量项的表结构信息，如下所示。

代码 3-6 CONSTANT_Utf8_info 表结构

```
CONSTANT_Utf8_info
{
    u1 tag;
    u2 length;
    u1 bytes[length];
}
```

上述代码示例中，tag 项表明了常量项的具体类型和格式，那么 CONSTANT_Utf8_info 常量项中 tag 项的值则为 CONSTANT_Utf8_info(1)。length 项指明了后续 bytes[]项的数组长度（字节数），而 bytes[]项是一个用于存储常量字符串信息的 byte 数组。在此大家需要注意，改进过的 UTF-8 编码格式与传统的 UTF-8 编码格式是存在一定区别的，改进过的 UTF-8 编码格式从‘\u0001’~‘\u007f’之间的字符使用 1 个字节来表示；从‘\u0080’~‘\u07ff’之间的字符使用 2 个字节来表示；而从‘\u0800’~‘\uffff’之间的字符则与传统的 UTF-8 编码格式一样使用 3 个字节来表示。关于传统的 UTF-8 编码格式的更多信息，大家可以参考《The Unicode Standard》（版本 6.0.0）中的 3.9 节“Unicode Encoding Forms”。

3.3.2 CONSTANT_Integer_info常量项

在常量池列表中，CONSTANT_Integer_info 常量项用于存储一个 int 类型的常量信息。在此大家需要注意，CONSTANT_Integer_info 常量项中仅仅只会存储 int 类型的值，并不会存储任何的符号引用信息。CONSTANT_Integer_info 常量项的表结构信息，如下所示：

代码 3-7 CONSTANT_Integer_info 表结构

```
CONSTANT_Integer_info
{
    u1 tag;
    u4 bytes;
}
```

上述代码示例中，CONSTANT_Integer_info 常量项中 tag 项的值为 CONSTANT_Integer_info(3)。由于 int 类型的二进制位数为 32 位，因此 bytes 项将会占用 4 个字节，并按照 big-endian 顺序去存储一个 int 类型的值。

3.3.3 CONSTANT_Float_info常量项

在常量池列表中，CONSTANT_Float_info 常量项用于存储一个 float 类型的常量信息。和 CONSTANT_Integer_info 常量项一样，CONSTANT_Float_info 常量项中同样也不会存储任何的符号引用信息，它仅仅只会用于存储 float 类型的值。CONSTANT_Float_info 常量项的表结构信息，如下所示：

代码 3-8 CONSTANT_Float_info 表结构

```
CONSTANT_Float_info
{
    u1 tag;
    u4 bytes;
}
```

相信细心的读者已经发现了，CONSTANT_Float_info 常量项的表结构与 CONSTANT_Integer_info 常量项的表结构其实是一模一样的。上述代码示例中，CONSTANT_Float_info 常量项中 tag 项的值为 CONSTANT_Integer_info(4)。由于 float 类型的二进制位数与 int 类型的二进制位数同样是 32 位，因此后续的 bytes 项同样也将占用 4 个字节，并按照 big-endian 顺序存储一个 float 类型的值。

3.3.4 CONSTANT_Long_info常量项

在常量池列表中，CONSTANT_Long_info 常量项用于存储一个 long 类型的常量信息。当然 CONSTANT_Long_info 常量项仅仅只会用于存储 long 类型的值，并不存储任何的符号引用信息。CONSTANT_Long_info 常量项的表结构信息，如下所示：

代码 3-9 CONSTANT_Long_info 表结构

```
CONSTANT_Long_info
{
    u1 tag;
    u8 bytes;
}
```

上述代码示例中，CONSTANT_Long_info 常量项中 tag 项的值为 CONSTANT_Integer_info(5)。由于 long 类型的二进制位数为 64 位，因此 bytes 项将会占用 8 个字节，并按照 big-endian 顺序存储一个 long 类型的值。

3.3.5 CONSTANT_Double_info常量项

在常量池列表中，CONSTANT_Double_info 常量项用于存储一个 double 类型的常量信息。和之前的 CONSTANT_Integer_info、CONSTANT_Float_info 以及 CONSTANT_Long_info 常量项一样，CONSTANT_Double_info 常量项并不存储任何的符号引用信息，仅仅只会用于存储 double 类型的值。CONSTANT_Double_info 常量项的表结构信息，如下所示：

代码 3-10 CONSTANT_Double_info 表结构

```
CONSTANT_Double_info
{
    u1 tag;
    u8 bytes;
}
```

上述代码示例中，CONSTANT_Double_info 常量项中 tag 项的值为 CONSTANT_Double_info(6)。由于 double 类型的二进制位数与 long 类型的二进制位数一样同样都是 64 位，因此后续的 bytes 项同样也将占用 8 个字节，并按照 big-endian 顺序去存储一个 double 类型的值。

3.3.6 CONSTANT_Class_info常量项

在常量池列表中，CONSTANT_Class_info 常量项用于存储一个指向常量池列表中 CONSTANT_Utf8_info 常量项（请阅读 3.3.1 节）的有效索引。CONSTANT_Class_info 常量项的表结构信息，如下所示：

代码 3-11 CONSTANT_Class_info 表结构

```
CONSTANT_Class_info
{
    u1 tag;
    u2 name_index;
}
```

上述代码示例中，CONSTANT_Class_info 常量项中 tag 项的值为 CONSTANT_Class_info(7)。而 name_index 则是一个指向常量池列表中一个类型为 CONSTANT_Utf8_info 常量项的有效索引，通过这个索引值即可成功获取 CONSTANT_Utf8_info 常量项中一个类或者接口的全限定名字符串。

JVM 中包含三种引用类型，分别是：类类型（class type）、数组类型（array type）和接口类型（interface type）。由此可见，数组同样也由对象来表示，那么 CONSTANT_Class_info 常量项必然也能够用于描述数组类型，即通过 name_index 项中的索引值指向 CONSTANT_Utf8_info 常量项来获取一个数组的描述符信息。关于数组类型，字段描述符“[C”所代表的含义就是一个 char 类型的一维数组，那么如果是一个 char 类型的二维数组则使用描述符“[[C”进行表示，依此类推。在此大家需要注意，由于 Java 语法规范规定了一个数组的最大维度只能够拥有 255 维，因此数组描述符最多也只能够包含 255 个描述符“[”。

3.3.7 CONSTANT_String_info常量项

在常量池列表中，CONSTANT_String_info 常量项用于存储一个指向常量池列表中 CONSTANT_Utf8_info 常量项（请阅读 3.3.1 节）的有效索引。CONSTANT_String_info 常量项的表结构信息，如下所示：

代码 3-12 CONSTANT_Class_info 表结构

```
CONSTANT_String_info
{
    u1 tag;
    u2 string_index;
}
```

上述代码示例中，CONSTANT_String_info 常量项中 tag 项的值为 CONSTANT_String_info(8)。而 name_index 则是一个指向常量池列表中类型为 CONSTANT_Utf8_info 常量项的有效索引，通过这个索引值即可成功获取 CONSTANT_Utf8_info 常量项中的文字字符串。

3.3.8 CONSTANT_Fieldref_info常量项

在常量池列表中，CONSTANT_Fieldref_info 常量项用于存储指向常量池列表中CONSTANT_Class_info 常量项（请阅读 3.3.6 节）和 CONSTANT_NameAndType_info 常量项（请阅读 3.3.11 节）的有效索引。CONSTANT_Fieldref_info 常量项的表结构信息，如下所示：

代码 3-13 CONSTANT_Fieldref_info 表结构

```
CONSTANT_Fieldref_info
{
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

上述代码示例中，CONSTANT_Fieldref_info 常量项中 tag 项的值为 CONSTANT_Fieldref_info(9)。class_index 项中的值则是一个指向常量池列表中 CONSTANT_Class_info 常量项的有效索引，通过这个索引值即可成功地确定当前字段所属的类或者接口。而 name_and_type_index 项中的值同样也是一个指向常量池列表中的索引，只不过该索引值指向的是 CONSTANT_TypeAndType_info 常量项，通过这个索引值即可成功获取到当前字段的简单名称和字段描述符。

3.3.9 CONSTANT_Methodref_info常量项

在常量池列表中，CONSTANT_Methodref_info 常量项和 CONSTANT_Fieldref_info 常量项一样，都用于存储指向常量池列表中 CONSTANT_Class_info 常量项（请阅读 3.3.6 节）和 CONSTANT_NameAndType_info 常量项（请阅读 3.3.11 节）的有效索引。CONSTANT_Methodref_info 常量项的表结构信息，如下所示：

代码 3-14 CONSTANT_Methodref_info 表结构

```
CONSTANT_Methodref_info
{
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

上述代码示例中，CONSTANT_Methodref_info 常量项中 tag 项的值为 CONSTANT_

Methodref_info(10)。class_index 项中的值则是一个指向常量池列表中 CONSTANT_Class_info 常量项的有效索引，通过这个索引值即可成功确定当前方法所属的类（不包括接口中的方法）。而 name_and_type_index 项中的值同样也是一个指向常量池列表中的索引，只不过该索引值指向的是 CONSTANT_TypeAndType_info 常量项，通过这个索引值即可成功获取当前方法的简单名称和方法描述符。

3.3.10 CONSTANT_InterfaceMethodref_info常量项

在常量池列表中，CONSTANT_InterfaceMethodref_info 常量项和 CONSTANT_Fieldref_info 以及 CONSTANT_Methodref_info 常量项一样，都用于存储指向常量池列表中 CONSTANT_Class_info 常量项（请阅读 3.3.6 节）和 CONSTANT_NameAndType_info 常量项（请阅读 3.3.11 节）的有效索引。CONSTANT_InterfaceMethodref_info 常量项的表结构信息，如下所示：

代码 3-15 CONSTANT_InterfaceMethodref_info 表结构

```
CONSTANT_InterfaceMethodref_info
{
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

上述代码示例中，CONSTANT_InterfaceMethodref_info 常量项中 tag 项的值为 CONSTANT_InterfaceMethodref_info(11)。class_index 项中的值则是一个指向常量池列表中 CONSTANT_Class_info 常量项的有效索引，通过这个索引值即可成功地确定当前方法所属的接口（不包括类中的方法）。而 name_and_type_index 项中的值同样也是一个指向常量池列表中的索引，只不过该索引值指向的是 CONSTANT_TypeAndType_info 常量项，通过这个索引值即可成功获取当前方法的简单名称和方法描述符。

3.3.11 CONSTANT_NameAndType_info常量项

在常量池列表中，CONSTANT_NameAndType_info 常量项用于存储指向常量池列表中 CONSTANT_Utf8_info 常量项（请阅读 3.3.1 节）的有效索引。CONSTANT_ NameAndType_info 常量项的表结构信息，如下所示：

代码 3-15 CONSTANT_NameAndType_info 表结构

```
CONSTANT_NameAndType_info
{
```

```
    u1 tag;
    u2 class_index;
    u2 descriptor_index;
}
```

上述代码示例中，CONSTANT_NameAndType_info 常量项中 tag 项的值为 CONSTANT_NameAndType_info(12)。class_index 项中的值则是一个指向常量池列表中 CONSTANT_Utf8_info 常量项的有效索引，通过这个索引值即可成功获取到当前字段或者方法的简单名称或者是一个特殊的方法名称<init>。而 descriptor_index 项中的值同样也是一个指向常量池列表中 CONSTANT_Utf8_info 常量项的索引，通过这个索引值即可成功获取到当前字段或者方法的描述符。

3.3.12 CONSTANT_MethodHandle_info常量项

在常量池列表中，CONSTANT_MethodHandle_info、CONSTANT_MethodType_info，以及 CONSTANT_InvokeDynamic_info 常量项是 Java7 中新增的 3 个常量项，JVM 的设计者们提供这 3 个常量项的目的是为了希望能够更好地支持动态语言的调用。本小节笔者优先从 CONSTANT_MethodHandle_info 常量项开始讲解，CONSTANT_MethodHandle_info 常量项主要用于表示方法句柄。CONSTANT_MethodHandle_info 常量项的表结构信息，如下所示：

代码 3-16 CONSTANT_MethodHandle_info 表结构

```
CONSTANT_MethodHandle_info
{
    u1 tag;
    u1 reference_kind;
    u2 reference_index;
}
```

上述代码示例中，CONSTANT_MethodHandle_info 常量项中 tag 项的值为 CONSTANT_MethodHandle_info（15）。reference_kind 项中的值必须在 1～9 之间，它决定了后续 reference_index 项中的方法句柄类型，方法句柄类型的值表示方法句柄的字节码行为。而 reference_index 项中的值则是一个指向常量池列表的有效索引，参考《Java 虚拟机规范 Java SE7 版》的描述来看，如果 reference_index 项的值为 1、2、3、4，那么 reference_index 项中的值则是一个指向常量池列表中 CONSTANT_Fieldref_info 常量项（请阅读 3.3.8 节）的索引，用于表示为一个字段创建的方法句柄；如果 reference_index 项的值为 5、6、7、7，那么 reference_index 项中的值则是一个指向常量池列表中 CONSTANT_Methodref_info 常量项（请阅读 3.3.9 节）的索引，用于表示为类的方法或者构造方法创建的方法句柄；如果

reference_index 项的值为 9，那么 reference_index 项中的值则是一个指向常量池列表中 CONSTANT_InterfaceMethodref_info 常量项（请阅读 3.3.10 节）的索引，用于表示为接口的方法创建的方法句柄。

3.3.13 CONSTANT_MethodType_info常量项

在常量池列表中，CONSTANT_MethodType_info 常量项用于表示方法的类型。CONSTANT_MethodType_info 常量项的表结构信息，如下所示：

代码 3-17 CONSTANT_MethodType_info 表结构

```
CONSTANT_MethodType_info
{
    u1 tag;
    u2 descriptor_index;
}
```

上述代码示例中，CONSTANT_MethodType_info 常量项中 tag 项的值为 CONSTANT_MethodType_info(16)。descriptor_index 项中的值则是一个指向常量池列表中 CONSTANT_Utf8_info 常量项（请阅读 3.3.1 节）的有效索引，用于表示方法描述符。

3.3.14 CONSTANT_InvokeDynamic_info常量项

在常量池列表中，CONSTANT_InvokeDynamic_info 常量项用于表示 invokedynamic 指令所用到的引用方法（Bootstrap Method）、引导方法使用动态调用名称（Dynamic Invocation Name）、参数和请求返回类型，以及有选择性的附加被称之为静态参数（Static Arguments）的常量序列。CONSTANT_InvokeDynamic_info 常量项的表结构信息，如下所示：

代码 3-18 CONSTANT_InvokeDynamic_info 表结构

```
CONSTANT_InvokeDynamic_info
{
    u1 tag;
    u2 bootstrap_method_attr_index;
    u2 name_and_type_index;
}
```

上述代码示例中，CONSTANT_InvokeDynamic_info 常量项中 tag 项的值为 CONSTANT_InvokeDynamic_info(18)。 bootstrap_method_attr_index 项的值必须是对当前字节码文件中引导方法的 bootstrap_method 数组的有效索引。而 name_and_type_index 项的值则是一个指向常量池列表中 CONSTANT_NameAndType_info 常量项（请阅读 3.3.11 节）的

有效索引，用于表示方法的简单名称和方法描述符。

3.4 字段表

相信大家都应该知道，在 Java 语法层面上是不允许在一个类或者接口中声明多个具有相同标示符名称的字段，当然如果这些字段彼此之间所处的作用域不相同时，那么这就会被允许。换句话说，Java 的设计者们之所以不允许这么做主要是为了避免一些不必要的异常情况出现，试想一下，如果相同的作用域上出现多个标示符名称相同的字段，那么你又如何有效访问到指定的目标字段上呢？因此 Java 语法规范不允许这么做。

在字节码文件中，每一个 field_info 项都对应着一个类或者接口中的字段信息，用于表示一个字段的完整信息，比如字段的标示符、访问修饰符（public、private 或 protected）、是类变量还是实例变量（static 修饰符）、是否是常量（final 修饰符）等。在此大家需要注意，由于存储在 field_info 项中的字段信息并不包括声明在方法内部或者代码块内的局部变量，因此多个变量之间的作用域就都是一样的，那么 Java 语法规范必然不允许在一个类或者接口中声明多个具有相同标示符名称的字段。但是和 Java 语法规范相反，字节码文件中却恰恰允许存放多个具有相同标示符名称的字段，唯一的条件就是这些字段之间的描述符不能相同。field_info 项的表结构信息，如下所示：

代码 3-19 field_info 表结构

```
field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

上述代码示例中，access_flags 项中的值用于存储在类或者接口中声明字段时所用到的访问修饰符。access_flags 项的标志信息，如表 3-5 所示。

表 3-5 access_flags 项的标志信息

标志名称	值	描述
ACC_PUBLIC	0x0001	声明字段的访问修饰符为 public
ACC_PRIVATE	0x0002	声明字段的访问修饰符为 private
ACC_PROTECTED	0x0004	声明字段的访问修饰符为 protected
ACC_STATIC	0x0008	将一个字段声明为类变量

续表

标志名称	值	描述
ACC_FINAL	0x0010	将一个字段声明为 final 类型
ACC_VOLATILE	0x0040	将一个字段声明为 volatile
ACC_TRANSIENT	0x0080	声明一个字段不允许被序列化
ACC_SYNTHETIC	0x1000	声明一个字段由编译器自动产生
ACC_ENUM	0x4000	声明一个枚举类型的字段

参考《Java 虚拟机规范（Java SE7 版）》的描述来看，在类中声明的字段，只能使用 ACC_PUBLIC、ACC_PRIVATE 以及 ACC_PROTECTED 这 3 个标志中的其中一个作为字段的访问修饰符去声明，而 ACC_FINAL 字段和 ACC_VOLATILE 字段不允许同时对同一个字段进行声明。而为接口中的字段声明访问修饰符时，只能够使用 ACC_PUBLIC、ACC_PRIVATE 及 ACC_PROTECTED 这 3 个标志中的其中一个。在此大家需要注意，如果 access_flags 项中没有用到的标志，一律要求设置为 0，并且 Java 虚拟机还必须忽略掉它。

name_index 项中的值则是一个指向常量池列表中 CONSTANT_Utf8_info（请阅读 3.3.1 节）常量项的有效索引，通过这个索引值即可成功获取当前字段的简单名称。而 descriptor_index 项中的值同样也是一个指向常量池列表中 CONSTANT_Utf8_info 常量项的有效索引，通过这个索引值即可成功获取当前字段的描述符。attributes_count 项指明了后续 attributes[]项的数组长度，attributes[]项中的每一个成员都必须是一个 attribute_info 结构的数据项（请阅读 3.6 节）。

3.5 方法表

相信大家都应该知道，在 Java 语法层面上是不允许在一个类或者接口中声明多个方法签名（方法参数、参数类型及参数个数）相同的方法，如果在程序中声明多个方法签名相同的方法时，前端编译器将无法正常执行编译。换句话说，Java 的设计者们之所以不允许这么做主要是为了避免一些不必要的异常情况出现，试想一下，如果一个类或者接口中出现多个方法签名相同的方法，那么你应该如何有效地访问到指定的目标方法上呢？因此 Java 语法规范不允许这么做。当然如果希望在一个类或者接口中声明多个名称相同的方法也并不是不可以，在面向对象（Object Oriented）的多态（*Polymorphism*）特性中，方法重载（Overload）正好可以满足这个场景，只不过唯一需要注意的是，如果希望实现方法重载，那么多个方法之间的方法签名一定不能相同。

在字节码文件中，每一个 method_info 项都对应着一个类或者接口中的方法信息，或者由编译器产生的方法信息（比如：类(接口)初始化方法<clinit>()和实例初始化方法<init>()），

用于表示一个方法的完整信息，比如方法标示符、方法的访问修饰符（public、private 或 protected）、方法的返回值类型以及方法的参数信息等。在此大家需要注意，尽管 Java 语法规范并不允许在一个类或者接口中声明多个方法签名相同的方法，但是和 Java 语法规范相反，字节码文件中却恰恰允许存放多个方法签名相同的方法，唯一的条件就是这些方法之间的返回值不能相同。method_info 项的表结构信息，如下所示：

代码 3-20　method_info 表结构

```
method_info {
        u2 access_flags;
        u2 name_index;
        u2 descriptor_index;
        u2 attributes_count;
        attribute_info attributes[attributes_count];
}
```

相信细心的读者已经发现了，method_info 项的表结构信息与 field_info 项的表结构信息几乎是一模一样的。上述代码示例中，access_flags 项中的值用于存储在类或者接口中声明方法时所用到的访问修饰符。access_flags 项的标志信息，如表 3-6 所示。

表 3-6　access_flags 项的标志信息

标志名称	值	描述
ACC_PUBLIC	0x0001	声明方法的访问修饰符为 public
ACC_PRIVATE	0x0002	声明方法的访问修饰符为 private
ACC_PROTECTED	0x0004	声明方法的访问修饰符为 protected
ACC_STATIC	0x0008	将一个方法声明为类方法
ACC_FINAL	0x0010	将一个方法声明为 final 方法
ACC_SYNCHRONIZED	0x0020	将一个方法声明为 synchronized 方法
ACC_BRIDGE	0x0040	将一个方法声明为 bridge 方法
ACC_VARARGS	0x0080	表示方法带有变长参数
ACC_NATIVE	0x0100	将一个方法声明为 native 方法
ACC_ABSTRACT	0x0400	将一个方法声明为 abstract 方法
ACC_STRICT	0x0800	将一个方法声明为 strictfp 方法，表示 FP-strict 浮点格式
ACC_SYNTHETIC	0x1000	声明为被表示的字段由编译器产生

参考《Java 虚拟机规范（Java SE7 版）》的描述来看，在类中声明的方法，只能够使用 ACC_PUBLIC、ACC_PRIVATE 以及 ACC_PROTECTED 这 3 个标志中的其中一个作为方法的访问修饰符去声明。如果设定了一个方法的 ACC_ABSTRACT 标志，那么它的 ACC_PRIVATE、ACC_STATIC、ACC_FINAL、ACC_SYNCHRONIZED、ACC_NATIVE 以

及 ACC_STRICT 标志都必须清除。而为接口中的方法声明访问修饰符时，只能够使用 ACC_PUBLIC 和 ACC_ABSTRACT 标志，除此之外的其他所有标志都不能使用，不过接口初始化方法<clinit>()可以使用 ACC_STRICT 标志。

在此大家需要注意，<init>()方法只允许使用 ACC_PUBLIC、ACC_PRIVATE 以及 ACC_PROTECTED 标志。而<clinit>()方法除了 ACC_STRICT 之外的所有标志都应该被忽略，因为任何 invoke 之类的字节码指令也无法调用<clinit>()方法，该方法只能够在类加载的过程中被 JVM 所调用。

name_index 项中的值则是一个指向常量池列表中 CONSTANT_Utf8_info（请阅读 3.3.1 节）常量项的有效索引，通过这个索引值即可成功获取到当前方法的简单名称。而 descriptor_index 项中的值同样也是一个指向常量池列表中 CONSTANT_Utf8_info 常量项的有效索引，通过这个索引值即可成功获取到当前方法的描述符。attributes_count 项指明了后续 attributes[]项的数组长度，attributes[]项中的每一个成员都必须是一个 attribute_info 结构的数据项（请阅读 3.6 节）。

3.6 属性表

属性在字节码文件中几乎随处可见，比如 ClassFile、field_info 项、method_info 项和 Code_attribute 项中都使用属性。属性表的通用格式，如下所示：

代码 3-21 attribute_info 表结构

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

上述代码示例中，attribute_name_index 项中的值是一个指向常量池列表中 CONSTANT_Utf8_info（请阅读 3.3.1 节）常量项的有效索引，通过这个索引值即可成功获取到当前属性的简单名称。attribute_length 项指明了后续 info[]项的数组长度（不过并不包括 attribute_name_index 项和 attribute_length 项在内的起始 6 个字节长度），info[]项中的成员并没有一个明确的要求一定要符合某种结构的数据项，也就是说，info[]项中的成员结构信息完全是自定义的，该项用于存储属性的数据信息。

参考《Java 虚拟机规范 Java SE7 版》的描述来看，目前一共定义约 21 种预定义属性，如表 3-7 所示。为了能够正确地解释字节码文件，Java 虚拟机规范一律要求所有的 Java 虚

拟机实现都必须能够识别出 Code、ConstantValue 和 Exceptions 这 3 个属性。除此之外，用于实现 Java 平台类库的 InnerClasses、EnclosingMethod 和 Synthetic 等另外 3 个属性也同样必须能够被 Java 虚拟机所识别。而关于其他一些预定义属性或者自定义属性（Java 虚拟机规范允许前端编译器在字节码文件的属性表中添加自定义属性，不过这些属性不能够与现有的属性名称重复），Java 虚拟机在运行时将需要判断是否能够有效识别，如果不能识别，那么就必须忽略掉它们。

表 3-7 当前预定义属性

属性名称	Java SE	版本号	使用位置
Code	1.0.2	45.3	method_info
ConstantValue	1.0.2	45.3	field_info
StackMapTable	6	50.0	Code_attribute
Exceptions	1.0.2	45.3	method_info
InnerClasses	1.1	45.3	ClassFile
EnclosingMethod	5.0	49.0	ClassFile
Synthetic	1.1	45.3	ClassFile、method_info、field_info
Signature	5.0	49.0	ClassFile、method_info、field_info
SourceFile	1.0.2	45.3	ClassFile
SourceDebugExtension	5.0	49.0	ClassFile
LineNumberTable	1.0.2	45.3	Code_attribute
LocalVariableTable	1.0.2	45.3	Code_attribute
LocalVariableTypeTable	5.0	49.0	ClassFile
Deprecated	1.1	45.3	ClassFile、method_info、field_info
RuntimeVisibleAnnotations	5.0	49.0	ClassFile、method_info、field_info
RuntimeInvisibleAnnotations	5.0	49.0	method_info
RuntimeVisibleParameterAnnotations	5.0	49.0	method_info
RuntimeInvisibleParameterAnnotations	5.0	49.0	method_info
AnnotationDefault	5.0	49.0	method_info
BootstrapMethods	7	51.0	ClassFile

3.6.1 Code属性

为了能够正确地解释字节码文件，Code 属性是 Java 虚拟机必须识别出的 3 个属性之一，Code 属性位于 mthod_info 项的属性表中。简单来说，一个 Code 属性主要用于存储方法的字节码指令以及其他一些辅助信息。在此大家需要注意，并非所有的 mthod_info 项中都必须包含 Code 属性，比如在类或者接口中使用 abstract 关键字声明的抽象方法和使用关键字 native 声明的本地方法就不包含 Code 属性，除此之外，其他任何的 mthod_info 项中都必须

包含 Code 属性。Code 属性的表结构信息，如下所示：

代码 3-22 Code_attribute 表结构

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    exception_info exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attribute_count];
}
```

上述代码示例中，attribute_name_index 项中的值是一个指向常量池列表中 CONSTANT_Utf8_info（请阅读 3.3.1 节）常量项的有效索引，通过这个索引值即可成功获取到当前属性的简单名称，即字符串“Code”。attribute_length 项指明了 Code 属性值的长度（不过并不包括 attribute_name_index 项和 attribute_length 项在内的起始 6 个字节长度）。max_stack 项指定了当前方法的操作数栈的最大深度。max_locals 项指定了当前方法的局部变量表（请阅读 8.1.1 节）所需的存储空间长度。code_length 项指明了后续 code[]项的数组长度，不过 code_length 的值必须大于 0，也就是说，code_length 项的值不能够为空，而 code[]项中的成员就是方法的字节码。exception_table_length 项指明了后续 exception_table[]项的数组长度，而 exception_table[]项其实是一个 exception_info 项的列表，每一个 exception_info 项都用于描述一个异常表项。attribute_count 项指明了后续 attributes[]项的数组长度，而 attributes[]项中的每一个成员都必须是一个 attribute_info 结构的数据项。

3.6.2 ConstantValue属性

和 Code 属性一样，ConstantValue 属性同样也属于 Java 虚拟机必须识别出的 3 个属性之一，ConstantValue 属性位于 field_info 项的属性表中。在一个 field_info 项的属性表中最多只允许包含一个 ConstantValue 属性，该属性主要用于通知 Java 虚拟机对代码中的类变量（这里并不包括实例变量）执行初始化操作。简单来说，类变量的初始化操作有两种方式：一种是由类（接口）初始化方法<clinit>()完成，而另一种则由 ConstantValue 属性完成。如果一个类变量被 final 和 static 这两个关键字修饰后，并且数据类型是原始类型或者 java.lang.String 类型的话，这个类变量就能够被 ConstantValue 属性执行初始化操作，反之一个类变量如果并没有被 final 关键字修饰，或者数据类型并非是原始类型或者 java.lang.String

类型，那么则只能由<clinit>()方法完成其初始化操作。关于<clinit>()方法究竟是如何对类变量执行初始化操作的，请阅读 7.2.5 节。

参考《Java 虚拟机规范（Java SE7 版）》的描述来看，Java 虚拟机只是强制要求了 field_info 表中的 access_flag 项必须设置有 ACC_STATIC 标志，而 ACC_FINAL 标志并非是必需的，它只是一个可选项，因为仅仅只是前端编译器对 final 关键字有要求而已。在此大家需要注意，如果 field_info 表中的 access_flag 项并没有被设置 ACC_STATIC 标志时，那么 Java 虚拟机将必须忽略掉它。ConstantValue 属性的表结构信息，如下所示：

代码 3-23　ConstantValue_attribute 表结构

```
ConstantValue_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 constantvalue_index;
}
```

上述代码示例中，attribute_name_index 项中的值是一个指向常量池列表中 CONSTANT_Utf8_info（请阅读 3.3.1 节）常量项的有效索引，通过这个索引值即可成功获取当前属性的简单名称，即字符串“ConstantValue”。attribute_length 项的值被永远固定为 2。而 constantvalue_index 项中的值则是一个指向常量池列表中 CONSTANT_Long_info（请阅读 3.3.4 节）、CONSTANT_Double_info（请阅读 3.3.5 节）、CONSTANT_Integer_info（请阅读 3.3.2 节）、CONSTANT_Float_info（请阅读 3.3.3 节）或者 CONSTANT_String_info（请阅读 3.3.7 节）常量项的有效索引。

3.6.3　Exceptions属性

和 Code 属性以及 ConstantValue 属性一样，Exceptions 属性同样也属于 Java 虚拟机必须识别出的 3 个属性之一，Exceptions 属性位于 method_info 项的属性表中。在一个 method_info 项的属性表中最多只允许包含一个 Exceptions 属性，该属性主要用于列举出当前方法通过关键字 throws 可能抛出的异常信息。Exceptions 属性的表结构信息，如下所示：

代码 3-24　Exceptions_attribute 表结构

```
Exceptions_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_exceptions;
    u2 exception_index_table[number_of_exceptions];
}
```

上述代码示例中，attribute_name_index 项中的值是一个指向常量池列表中 CONSTANT_Utf8_info（请阅读 3.3.1 节）常量项的有效索引，通过这个索引值即可成功获取当前属性的简单名称，即字符串“Exceptions”。attribute_length 项指明了 Exceptions 属性值的长度（不过并不包括 attribute_name_index 项和 attribute_length 项在内的起始 6 个字节的长度）。number_of_exceptions 项则指明了后续 exception_index_table[]项的数组长度，而 exception_index_table[] 项中的每一个成员都必须是一个指向常量池列表中 CONSTANT_Class_info（请阅读 3.3.6 节）常量项的有效索引，通过这个索引值即可成功获取到当前方法通过关键字 throws 可能抛出的异常信息。

3.6.4 LineNumberTable属性

相信大家在程序出现异常的时候，经常需要根据错误异常日志中给定的错误行号以及错误代码所属的文件名称来分析和解决问题，那么如果希望在程序出现异常的时候，堆栈信息中不输出任何的错误行号以及错误代码所属的文件名称，大家则可以在使用前端编译器将 Java 代码编译为字节码时设定选项“-g:none”即可。如下所示：

代码 3-25 设置选项“-g:none”

```
Exception in thread "main" java.lang.ArithmeticException: / by zeroat Test.main(Unknown
Source)
```

如果在编译时设定有选项“-g:none”时，那么程序中一旦出现异常时，堆栈信息中将再也不会输出任何的错误行号以及错误代码所属的文件名称。简单来说，LineNumberTable 属性就用于描述 Java 代码中的行号与字节码文件中的字节码行号之间的对应关系，LineNumberTable 是一个可选属性，它位于 Code_attribute 项的属性表中。LineNumberTable 属性的表结构信息，如下所示：

代码 3-26 LineNumberTable_attribute 表结构

```
LineNumberTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 line_number_table_length;
    line_number_info line_number_table;
}
```

上述代码示例中，attribute_name_index 项中的值是一个指向常量池列表中 CONSTANT_Utf8_info（请阅读 3.3.1 节）常量项的有效索引，通过这个索引值即可成功获取到当前属性的简单名称，即字符串“LineNumberTable”。attribute_length 项指明了 LineNumberTable 属性值的长度（不过并不包括 attribute_name_index 项和 attribute_length 项

在内的起始 6 个字节的长度）。line_number_table_length 项指明了后续 line_number_table[]项的数组长度，而 line_number_table[]项中的每个成员都包括了 start_pc 和 line_number 这两个 u2 类型的数据项，start_pc 用于表示字节码文件中的字节码行号，而 line_number 则用于表示 Java 代码中的行号。

3.6.5 SourceFile属性

和 LineNumberTable 属性一样，SourceFile 属性同样也是一个可选属性，该属性位于 ClassFile 的属性表中，用于记录生成字节码文件的源文件名称。如果在编译时设定有选项“-g:none”时，那么程序中一旦出现异常时，堆栈信息中将再也不会输出任何的错误行号以及错误代码所属的文件名称，如代码 3-25 所示。SourceFile 属性的表结构信息，如下所示：

代码 3-26 SourceFile_attribute 表结构

```
SourceFile_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 sourcefile_index;
}
```

上述代码示例中，attribute_name_index 项中的值是一个指向常量池列表中 CONSTANT_Utf8_info（请阅读 3.3.1 节）常量项的有效索引，通过这个索引值即可成功获取到当前属性的简单名称，即字符串“SourceFile”。attribute_length 项的值被永远固定为 2。而 sourcefile_index 项中的值是一个指向常量池列表中 CONSTANT_Utf8_info（请阅读 3.3.1 节）常量项的有效索引，通过这个索引值即可成功获取源文件的文件名称。

3.6.6 LocalVariableTable属性

LocalVariableTable 属性用于描述局部变量表（请阅读 8.1.1 节）中的局部变量与 Java 代码中定义的变量之间的对应关系，LocalVariableTable 是一个可选属性，该属性位于 Code_attribute 项的属性表中。LocalVariableTable 属性的表结构信息，如下所示：

代码 3-27 LocalVariableTable_attribute 表结构

```
LocalVariableTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 local_variable_table_length;
    local_variable_info local_variable_table;
}
```

上述代码示例中，attribute_name_index 项中的值是一个指向常量池列表中 CONSTANT_Utf8_info（请阅读 3.3.1 节）常量项的有效索引，通过这个索引值即可成功获取当前属性的简单名称，即字符串“LocalVariableTable”。attribute_length 项指明了 Code 属性值的长度（不过并不包括 attribute_name_index 项和 attribute_length 项在内的起始 6 个字节长度）。local_variable_table_length 项指明了后续 local_variable_table[]项的数组长度，而 local_variable_table[]项中的每个成员都包括了 start_pc、length、name_index、descriptor_index 和 index 等 5 个 u2 类型的数据项。start_pc 和 length 项用于表示局部变量的生命周期开始的字节码偏移量及其作用范围覆盖的长度，start_pc+length 就是局部变量在字节码文件中的作用域范围；name_index 和 descriptor_index 项中的值同样都是一个指向常量池列表中 CONSTANT_Utf8_info 常量项的有效索引，通过这个索引值即可成功获取到当前局部变量的简单名称和描述符；而 index 项用于表示局部变量在局部变量表中 Slot（变量槽）的位置。

3.6.7 InnerClasses属性

InnerClasses 属性用于描述内部类与宿主类之间的关联关系，该属性位于 ClassFile 的属性表中。简单来说，如果在一个宿主类中定义内部类时，那么前端编译器将会为它及其所包含的内部类生成 InnerClasses 属性。InnerClasses 属性的表结构信息，如下所示：

代码 3-28 InnerClasses_attribute 表结构

```
InnerClasses_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_classes;
    inner_classes_info inner_classes;
}
```

上述代码示例中，attribute_name_index 项中的值是一个指向常量池列表中 CONSTANT_Utf8_info（请阅读 3.3.1 节）常量项的有效索引，通过这个索引值即可成功获取到当前属性的简单名称，即字符串“InnerClasses”。attribute_length 项指明了 Code 属性值的长度（不过并不包括 attribute_name_index 项和 attribute_length 项在内的起始 6 个字节的长度）。number_of_classes 项指明了后续 inner_classes[]项的数组长度，也就是一个类中究竟包含多少个内部类，而 inner_classes[]项中的每个成员都是一个 inner_classes_info 项的数据项，用于表示内部类信息。

3.6.8 BootstrapMethods属性

如表 3-7 所示，BootstrapMethods 属性是在 Java7 之后新增到字节码规范中的，该属性位于 ClassFile 的属性表中，用于保存 invokedynamic 指令引用的引导方法限定符。据《Java 虚拟机规范（Java SE7 版）》的描述来看，如果某个给定 ClassFile 结构的常量池中至少有一个 CONSTANT_InvokeDynamic_info 常量项（请阅读 3.3.14 节），那么这个 ClassFile 结构的属性表中必须有一个明确的 BootstrapMethods 属性，并且 ClassFile 结构的属性表中最多只能有一个 BootstrapMethods 属性。BootstrapMethods 属性的表结构信息，如下所示：

代码 3-29 BootstrapMethods_attribute 表结构

```
BootstrapMethods_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 num_bootstrap_methods;
    bootstrap_method bootstrap_methods;
}
```

上述代码示例中，attribute_name_index 项中的值是一个指向常量池列表中 CONSTANT_Utf8_info（请阅读 3.3.1 节）常量项的有效索引，通过这个索引值即可成功获取到当前属性的简单名称，即字符串“BootstrapMethods”。attribute_length 项指明了 Code 属性值的长度（不过并不包括 attribute_name_index 项和 attribute_length 项在内的起始 6 个字节长度）。num_bootstrap_methods 项指明了后续 bootstrap_methods[]项的数组长度，而 bootstrap_methods[]项中的值是一个指向常量池列表中 CONSTANT_MethodHandle_info 常量项（请阅读 3.3.12 节）常量项的有效索引，它代表了一个引导方法，还包含了这个引导方法静态参数的序列（可能为空）。bootstrap_methods[] 项中的每个成员都包括了 bootstrap_method_ref、num_bootstrap_arguments 和 bootstrap_arguments 这三个 u2 类型的数据项。其中 bootstrap_method_ref 项中的值是一个指向常量池列表中 CONSTANT_MethodHandle_info 常量项（请阅读 3.3.12 节）常量项的有效索引；num_bootstrap_arguments 项指明了后续 bootstrap_arguments[]项的数组长度，而 bootstrap_arguments[]项中的值则是一个指向常量池列表中 CONSTANT_Long_info（请阅读 3.3.4 节）、CONSTANT_Double_info（请阅读 3.3.5 节）、CONSTANT_Integer_info（请阅读 3.3.2 节）、CONSTANT_Float_info（请阅读 3.3.3 节）、CONSTANT_String_info（请阅读 3.3.7 节）、CONSTANT_Class_info（请阅读 3.3.6 节）、CONSTANT_MethodHandle_info（请阅读 3.3.12 节）或者 CONSTANT_MethodType_info（请阅读 3.3.13 节）常量项的有效索引。

3.7 本章小结

本章笔者详细地讲解了常量池表、字段表、方法表及属性表等字节码文件的内部组成结构。由于字节码文件是 Java 程序实现跨平台的基石，当大家对字节码文件的内部组成结构有了一定的了解之后，在接下来的后续章节中，我们将会以运行时的角度去看待字节码指令是如何在 Java 虚拟机中执行的。

第 4 章

剖析HotSpot的Launcher

JVM 只有由启动器成功启动之后，才能够正常运行，而在 HotSpot 中 Launcher 就承担着这个任务。对于绝大多数的 Java 开发人员而言，或许并不是很清楚 HotSpot 的启动过程与细节，那么从本章开始，笔者将会带领大家一起走进 HotSpot 的源码世界中去一探究竟。

4.1 HotSpot的源码目录结构

在开始正式深入学习 HotSpot 的源码之前，大家首先需要明白 HotSpot 的源码目录结构是怎样构成的，以及每一个目录中所包含的特定功能模块实现是什么。只有在彻底弄清楚这些问题后，才能更有针对性地阅读和学习 HotSpot 的源码。尽管 HotSpot 只是属于 OpenJDK 项目中的一个功能子集，但是 HotSpot 却在整个 OpenJDK 项目中占据了非常重要的地位；所以本书只会针对 HotSpot 项目的源码进行讲解，大家只需关注 HotSpot 即可。

在解压后的 HotSpot 源码目录下，主要由 agent、make、src 和 test 这 4 个子目录构成了 HotSpot 源码的整体目录结构。其中 agent 目录下包含了 Serviceability Agent 的客户端实现；make 目录下包含了用于 build 出 HotSpot 的各种配置文件；而 src 目录则是其中最重要的一个子目录，该目录下包含了 HotSpot 的所有源码实现（比如：与 CPU 的相关实现、与操作系统的相关实现、与平台无关的通用实现、HotSpot 的核心功能实现、各类 GC 的实现，以及执行引擎的相关实现等）；至于 test 目录下仅仅只是包含了 HotSpot 相关的一些单元测试用例。

在此需要提醒大家，尽管 Java 继承了 C 语言的语法结构，并改编了 C++语言的对象模型，但是彼此之间仍然存在着一定的语法差异。所以 Java 开发人员在阅读 HotSpot 的源码时，千万不要停留在语法细节上钻牛角尖，只需要关注具体的数据结构和算法即可。

HotSpot的源码目录结构，如下所示[①]：

```
├─agent                    Serviceability Agent 的客户端实现
├─make                     用于 build 出 HotSpot 的各种配置文件
├─src                      HotSpot 的所有源代码
│  ├─cpu                       CPU 相关代码
│  ├─os                        操作系统相关代码
│  ├─os_cpu                    操作系统+CPU 的组合相关的代码
│  └─share                     平台无关的共通代码
│      ├─tools                     工具
│      │  ├─hsdis                        反汇编插件
│      │  ├─IdealGraphVisualizer         将 server 编译器的中间代码可视化的工具
│      │  ├─launcher                     启动程序"java"
│      │  ├─LogCompilation               将-XX:+LogCompilation 输出的日志(hotspot.
                                           log)整理成更容易阅读的格式的工具
│      │  └─ProjectCreator               生成 Visual Studio 的 project 文件的工具
│      └─vm                        HotSpot VM 的核心代码
│          ├─adlc                        平台描述文件(上面的 cpu 或 os_cpu 里的*.ad
                                           文件)的编译器
│          ├─asm                         汇编器接口
│          ├─c1                          client 编译器（又称"C1"）
│          ├─ci                          动态编译器的公共服务/从动态编译器到 VM 的接口
│          ├─classfile                   类文件的处理（包括类加载和系统符号表等）
│          ├─code                        动态生成的代码的管理
│          ├─compiler                    从 VM 调用动态编译器的接口
│          ├─gc_implementation           GC 的实现
│          │  ├─concurrentMarkSweep          Concurrent Mark Sweep GC 的实现
│          │  ├─g1                           Garbage-First GC 的实现
                                               （不使用老的分代式 GC 框架）
│          │  ├─parallelScavenge             ParallelScavenge GC 的实现(server
                                               VM 默认不使用老的分代式 GC 框架)
│          │  ├─parNew                       ParNew GC 的实现
│          │  └─shared                       GC 的共通实现
│          ├─gc_interface                GC 的接口
│          ├─interpreter                 解释器，包括“模板解释器”（官方版在用）和
                                           “C++解释器”(官方版不在用)
│          ├─libadt                      一些抽象数据结构
│          ├─memory                      内存管理相关（老的分代式 GC 框架也在这里）
│          ├─oops                        HotSpot VM 的对象系统的实现
│          ├─opto                        server 编译器（又称"C2"或"Opto"）
│          ├─prims                       HotSpot VM 的对外接口，包括部分标准库的
                                           native 部分和 JVMTI 实现
│          ├─runtime                     运行时支持库(包括线程管理、编译器调度、锁、
```

① HotSpot 的源码目录结构引用自博文：http://hllvm.group.iteye.com/group/topic/26998。

```
                                    反射等)
│          ├─services               主要是用来支持 JMX 之类的管理功能的接口
│          ├─shark                  基于 LLVM 的 JIT 编译器（官方版里没有使用）
│          └─utilities              一些基本的工具类
└─test                          单元测试
```

只有当大家弄清楚 HotSpot 的源码目录结构后，才能更快地熟悉 HotSpot 中每一个目录下所包含的特定功能模块实现。

4.2 Launcher简介

Launcher 是一种用于启动 JVM 进程的启动器，并且可以根据类别划分为两种不同的 Launcher，一种是正式版的启动器，也就是大家在 Windows 平台下经常用到的 java.exe 和 javaw.exe 程序。前者在运行时会保留控制台，以及显示程序的输出信息。而后者主要是用于执行 Java 的 GUI 程序，也就是说，使用 javaw.exe 执行 Java 程序时将不会显示任何的程序的输出信息。关于 Launcher 的具体使用方式和标准选项配置，大家可以在控制台中输入命令“java-help”，如下所示。

具体用法：

```
java [-options] class [args...](执行类)
```

或

```
java [-options] -jar jarfile [args...](执行 jar 文件)
```

其中选项包括：

```
-d32                                使用 32 位数据模型(如果可用)
-d64                                使用 64 位数据模型(如果可用)
-client                             选择"client"VM
-server                             选择"server"VM
-hotspot                            是"client"VM 的同义词[已过时], 默认 VM 是
                                    client
-cp                                 目录和 zip/jar 文件的类搜索路径
-classpath                          目录和 zip/jar 文件的类搜索路径用“;”分隔的
                                    目录,JAR 档案和 ZIP 档案列表, 用于搜索类文件
-D<name>=<value>                    设置系统属性
-verbose[:class|gc|jni]             启用详细输出
-version                            输出产品版本并退出
-version:<value>                    需要指定的版本才能运行
```

-showversion	输出产品版本并继续
-jre-restrict-search|-no-jre-restrict-search	在版本搜索中包括/排除用户专用 JRE
-? -help	输出此帮助消息
-X	输出非标准选项的帮助
-ea[:<packagename>...|:<classname>]	
-enableassertions[:<packagename>...|:<classname>]	按指定的粒度启用断言
-disableassertions[:<packagename>...|:<classname>]	禁用具有指定粒度的断言
-esa | -enablesystemassertions	启用系统断言
-dsa | -disablesystemassertions	禁用系统断言
-agentlib:<libname>[=<options>]	加载本机代理库<libname>,例如-agentlib:hprof, 另请参阅-agentlib:jdwp=help 和-agentlib:hprof=help
-agentpath:<pathname>[=<options>]	按完整路径名加载本机代理库
-javaagent:<jarpath>[=<options>]	加载 Java 编程语言代理, 请参阅 java.lang.instrument
-splash:<imagepath>	使用指定的图像显示启动屏幕

大家千万不要认为 Launcher 就是虚拟机实现，其实从严格意义上来说，Launcher 只是一个封装了虚拟机的执行外壳，由它负责装载 JRE 环境和 Windows 平台下的 jvm.dll 动态链接库（Linux 平台下则是装载 libjvm.so）。在一个 JVM 的进程内部，只能执行一个指定的 Java 程序，也就是说，当执行多个 Java 程序时，也就意味着同时启动了多个 JVM 进程。在 1.5.6 节中，本书示例了如何编译一个 Debug 版本的 HotSpot，所以为了调试跟踪方面，大家可以使用 Java 的另一种启动器 gamma。在 HotSpot 中 Launcher 是使用 C 语言编写的，对比 gamma 和 java 后不难发现两者的源码几乎是一模一样的，仅存在少量差异，也就是说，在 OpenJDK 中 gamma 和 java 是共用的同一套 Launcher 源码实现。gamma 的源码在 HotSpot 的源码目录下，大家可以在/hotspot/src/share/tools/launcher/java.c 中找到。而 java 却并非包含在 HotSpot 的源码目录下，而是包含在/jdk/src/share/bin/main.c 中。

尽管 Launcher 并非是 HotSpot 的核心，甚至应该算是 HotSpot 中比较“外围”的功能模块，但既然是这样，笔者为什么还需要大费周章地对 Launcher 的源码进行剖析？其实了解 Launcher 的执行原理是非常有意义的。既然 Launcher 是 JVM 的启动器，那么必然会由它负责调用 HotSpot 的核心代码对 JVM 执行初始化，以及由它负责维护 JVM 的整个生命周期，所以理解 Launcher 的执行原理，是迈进 HotSpot 的第一步。

4.3 跟踪Launcher的执行过程

本书并非只是一本单纯讲解 Java 虚拟机原理的理论性读物，从本章开始，大家将会从以往的枯燥和乏味中深入到 HotSpot 的具体实现细节上。对于那些曾经想要深入研究 JVM 技术却又止步于源码面前的 Java 开发人员而言，本书的知识点将会是你们迫切想要得到的答案。

尽管每一个 Java 开发人员对 Launcher 的使用都非常熟悉，但这并不代表对 Launcher 的执行过程也了如指掌，所以本章将会重点讲解关于 Launcher 的执行过程并剖析源码细节。由于 HotSpot 的 Launcher 是采用 C 语言编写的，所以对于具备一定 C 语言功底的读者阅读，必然是最好不过的。但如果你仅仅只是专攻于 Java 技术，也并非无法理解本章的一些源码示例，毕竟 Java 的语法结构继承自 C 语言，所以对于一些简单 C 语法相信你也一定能够理解。在正式开始讲解之前，笔者还是需要再次重申一次关于源码的阅读方式，希望大家千万不要过多停留在语法细节上，只需关注具体的数据结构和算法即可。

4.3.1 使用Launcher启动JVM

Launcher 从启动到结束的整个执行过程，如图 4-1 所示。当成功启动 Launcher 后，首先进入到 Launcher 的启动函数中，这一点和 Java 程序一样，Launcher 的启动函数同样也是 main()。main()函数的主要任务是负责创建运行环境，以及启动一个全新的线程去执行 JVM 的初始化和调用 Java 程序的 main()方法。

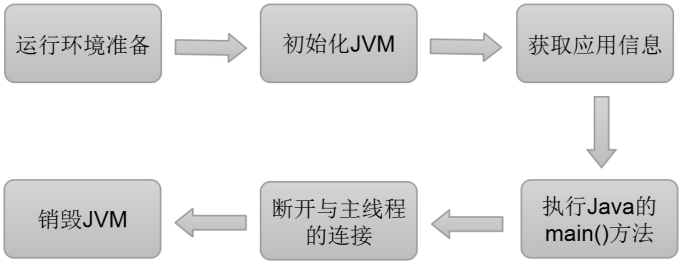


图 4-1 Launcher 的执行过程

当 main()函数成功创建运行后，就会启动一个全新的线程去调用 JavaMain()函数，而 JavaMain()函数的主要任务是负责调用 InitializeJVM()函数。顾名思义，InitializeJVM()函数肯定会负责 JVM 初始化的相关工作，但 InitializeJVM()函数本身却并不具备初始化 JVM 的能力，而是由它调用本地函数 JNI_CreateJavaVM()去完成真正意义上的 JVM 初始化。

当 JVM 初始化完成后，Launcher 接着调用 LoadClass()函数和 GetStaticMethodId()函数，分别获取 Java 程序的启动类和启动方法。当执行完这两个步骤后，Launcher 就会调用本地

函数 jni_CallStaticVoidMethod()执行 Java 程序的 main()方法。

最后 Launcher 还会调用本地函数 jni_DetachCurrentThread()断开与主线程的连接。当成功与主线程断开连接后，Launcher 就会一直等待程序中所有的非守护线程（non-daemon thread）全部执行结束，然后调用本地函数 jni_DestroyJavaVM()对 JVM 执行销毁。在此需要提醒大家，在 JDK1.2 版本之前，只有主线程才允许对 JVM 执行销毁，而在 JDK1.2 及后续版本中则没有此限制，非主线程也允许对 JVM 执行销毁。

4.3.2 启动函数main()

当 Launcher 成功启动后，首先会进入到 main()函数中对与运行环境相关的局部变量进行初始化。初始化后的局部变量不仅在程序后续创建运行环境时需要用到，在调用 JavaMain()函数时，也需要将这些变量传递过去，如下所示：

代码 4-1 初始化与运行环境相关的局部变量

```
/* 初始化与运行环境相关的局部变量 */
char *jarfile = 0;
char *classname = 0;
char *s = 0;
char *main_class = NULL;
int ret;
InvocationFunctions ifn;
jlong start, end;
char jrepath[MAXPATHLEN], jvmpath[MAXPATHLEN];
char ** original_argv = argv;
```

当完成对相关局部变量的初始化工作后，main()函数下一步则会创建运行环境，如下所示：

代码 4-2 创建运行环境

```
/* 创建运行环境 */
CreateExecutionEnvironment(&argc, &argv,
                           jrepath, sizeof(jrepath),
                           jvmpath, sizeof(jvmpath),
                           original_argv);
printf("Using java runtime at: %s\n", jrepath);
```

如果这一系列的前期准备工作都能够顺利执行完成的话，Launcher 就会执行到 main()函数的程序末尾，启动一个新线程作为主线程去调用 JavaMain()函数执行 JVM 初始化等相关工作。在此大家需要注意，一旦在 main()函数中启动了新线程，Launcher 就会阻塞原有线程，如下所示：

代码 4-3　创建一个新线程来执行后续任务

```
/* 创建一个新的线程去执行 JVM 的初始化和调用 Java 程序的 main()方法 */
struct JavaMainArgs args;
args.argc = argc;
args.argv = argv;
args.jarfile = jarfile;
args.classname = classname;
args.ifn = ifn;
return ContinueInNewThread(JavaMain, threadStackSize, (void*)&args);
```

为了使大家阅读更方便，本书示例了 main()函数的完整代码，如下所示：

代码 4-4　启动函数 main()的完整代码

```
int main(int argc, char ** argv)
{
    /* 初始化与运行环境相关的局部变量 */
    char *jarfile = 0;
    char *classname = 0;
    char *s = 0;
    char *main_class = NULL;
    int ret;
    InvocationFunctions ifn;
    jlong start, end;
    char jrepath[MAXPATHLEN], jvmpath[MAXPATHLEN];
    char ** original_argv = argv;

    if (getenv("_JAVA_LAUNCHER_DEBUG") != 0) {
        _launcher_debug = JNI_TRUE;
        printf("----_JAVA_LAUNCHER_DEBUG----\n");
    }

#ifndef GAMMA
    /*
     * Make sure the specified version of the JRE is running.
     *
     * There are three things to note about the SelectVersion() routine:
     *  1) If the version running isn't correct, this routine doesn't
     *     return (either the correct version has been exec'd or an error
     *     was issued).
     *  2) Argc and Argv in this scope are *not* altered by this routine.
     *     It is the responsibility of subsequent code to ignore the
     *     arguments handled by this routine.
     *  3) As a side-effect, the variable "main_class" is guaranteed to
     *     be set (if it should ever be set).  This isn't exactly the
     *     poster child for structured programming, but it is a small
     *     price to pay for not processing a jar file operand twice.
     *     (Note: This side effect has been disabled.  See comment on
```

```
     *     bugid 5030265 below.)
     */
    SelectVersion(argc, argv, &main_class);
#endif /* ifndef GAMMA */

    /* copy original argv */
    {
      int i;
      original_argv = (char**)JLI_MemAlloc(sizeof(char*)*(argc+1));
      for(i = 0; i < argc+1; i++)
        original_argv[i] = argv[i];
    }

    /* 创建运行环境 */
    CreateExecutionEnvironment(&argc, &argv,
                               jrepath, sizeof(jrepath),
                               jvmpath, sizeof(jvmpath),
                               original_argv);
    printf("Using java runtime at: %s\n", jrepath);
    ifn.CreateJavaVM = 0;
    ifn.GetDefaultJavaVMInitArgs = 0;

    if (_launcher_debug)
      start = CounterGet();
    if (!LoadJavaVM(jvmpath, &ifn)) {
      exit(6);
    }
    if (_launcher_debug) {
      end   = CounterGet();
      printf("%ld micro seconds to LoadJavaVM\n",
             (long)(jint)Counter2Micros(end-start));
    }

#ifdef JAVA_ARGS  /* javac, jar and friends. */
    progname = "java";
#else             /* java, oldjava, javaw and friends */
#ifdef PROGNAME
    progname = PROGNAME;
#else
    progname = *argv;
    if ((s = strrchr(progname, FILE_SEPARATOR)) != 0) {
        progname = s + 1;
    }
#endif /* PROGNAME */
#endif /* JAVA_ARGS */
    ++argv;
    --argc;

#ifdef JAVA_ARGS
```

```
    /* Preprocess wrapper arguments */
    TranslateApplicationArgs(&argc, &argv);
    if (!AddApplicationOptions()) {
        exit(1);
    }
#endif

    /* Set default CLASSPATH */
    if ((s = getenv("CLASSPATH")) == 0) {
        s = ".";
    }
#ifndef JAVA_ARGS
    SetClassPath(s);
#endif

    /*
     *  Parse command line options; if the return value of
     *  ParseArguments is false, the program should exit.
     */
    if (!ParseArguments(&argc, &argv, &jarfile, &classname, &ret, jvmpath)) {
      exit(ret);
    }

    /* Override class path if -jar flag was specified */
    if (jarfile != 0) {
        SetClassPath(jarfile);
    }

    /* set the -Dsun.java.command pseudo property */
    SetJavaCommandLineProp(classname, jarfile, argc, argv);

    /* Set the -Dsun.java.launcher pseudo property */
    SetJavaLauncherProp();

    /* set the -Dsun.java.launcher.* platform properties */
    SetJavaLauncherPlatformProps();

#ifndef GAMMA
    /* Show the splash screen if needed */
    ShowSplashScreen();
#endif

    /*
     * Done with all command line processing and potential re-execs so
     * clean up the environment.
     */
    (void)UnsetEnv(ENV_ENTRY);
#ifndef GAMMA
    (void)UnsetEnv(SPLASH_FILE_ENV_ENTRY);
    (void)UnsetEnv(SPLASH_JAR_ENV_ENTRY);
```

```
    JLI_MemFree(splash_jar_entry);
    JLI_MemFree(splash_file_entry);
#endif

    /*
     * If user doesn't specify stack size, check if VM has a preference.
     * Note that HotSpot no longer supports JNI_VERSION_1_1 but it will
     * return its default stack size through the init args structure.
     */
    if (threadStackSize == 0) {
      struct JDK1_1InitArgs args1_1;
      memset((void*)&args1_1, 0, sizeof(args1_1));
      args1_1.version = JNI_VERSION_1_1;
      ifn.GetDefaultJavaVMInitArgs(&args1_1);  /* ignore return value */
      if (args1_1.javaStackSize > 0) {
         threadStackSize = args1_1.javaStackSize;
      }
    }
    {
/* 创建一个新的线程去执行 JVM 的初始化和调用 Java 程序的 main()方法 */
      struct JavaMainArgs args;
      args.argc = argc;
      args.argv = argv;
      args.jarfile = jarfile;
      args.classname = classname;
      args.ifn = ifn;
      return ContinueInNewThread(JavaMain, threadStackSize, (void*)&args);
    }
}
```

4.3.3 在主线程中执行JavaMain()函数

从严格意义上来说，基本上当 main()函数完成运行环境的创建后，使命就会全部交给在主线程中执行的 JavaMain()函数。JavaMain()函数首先会接收从 main()函数中传递过来的一些变量参数，然后初始化几个比较重要的局部变量，如下所示：

代码 4-5 接受参数并初始化一些局部变量

```
/* 接收 main()函数中传递过来的变量参数 */
struct JavaMainArgs *args = (struct JavaMainArgs *)_args;

int argc = args->argc;
char **argv = args->argv;
char *jarfile = args->jarfile;
char *classname = args->classname;
InvocationFunctions ifn = args->ifn;
```

```
/* 初始化比较重要的一些局部变量 */
JavaVM *vm = 0;
JNIEnv *env = 0;
jstring mainClassName;
jclass mainClass;
jmethodID mainID;
jobjectArray mainArgs;
int ret = 0;
jlong start, end;
```

如果大家细心的话，应该会发现在上述语句中出现了几个非基本数据类型的数据结构，比如JavaMainArgs、JavaVM、JNIEnv和InvocationFunctions等类型。其实在C语言中，我们可以将这种类型的数据结构称为结构体。那么所谓结构体，简单来说就是通过使用struct关键字修饰并由不同的数据类型或函数指针[②] 组成的一种特殊数据结构。尽管结构体类似于数组类型的数据结构，但结构体和数组还是存在一些本质的差别。首先结构体中允许声明多种数据类型，这一点和数组中的单数据类型不同，其次相同结构的结构体变量还可以实现相互赋值。

当大家理解什么是结构体后，再来看看包含在结构体中的函数指针具体负责哪些任务。其实这些函数指针主要是应用在初始化 JVM、调用 Java 程序的 main()方法、断开主线程连接和执行 JVM 销毁等功能上。

JavaMainArgs 类型中的数据类型早在 main()函数启动新线程调用 JavaMain()函数时就已经指定好了。JavaMainArgs 类型的组成结构，如下所示：

代码 4-6　JavaMainArgs 类的组成结构

```
struct JavaMainArgs {
  int    argc;
  char ** argv;
  char *  jarfile;
  char *  classname;
  InvocationFunctions ifn;};
```

JavaMainArgs 类型中除声明基本数据类型外，还声明 InvocationFunctions 类型。InvocationFunctions 类型其实就是一个拥有少量函数指针的结构体，和 JavaMainArgs 类型一样，InvocationFunctions 在 JavaMain()函数执行之前就已经指定好了其中的值。InvocationFunctions 类型中包含的函数指针与对应的目标函数，如下所示：

□ CreateJavaVM --> JNI_CreateJavaVM

② 函数指针：结构体中包含的一种用于指向目标函数的指针。

□ GetDefaultJavaVMInitArgs --> JNI_GetDefaultJavaVMInitArgs

JavaVM 类型中包含的函数指针所指向的目标函数主要是用于完成断开主线程连接和执行 JVM 销毁等功能，其中函数指针的值由 InitializeJVM()函数在运行时指定。JavaVM 类型中包含的函数指针与对应的目标函数，如下所示：

□ DetachCurrentThread --> jni_DetachCurrentThread
□ DestroyJavaVM --> jni_DestroyJavaVM

由于 JNIEnv 类型中包含的函数指针较多，本书就不再一一列举了。当成功将函数指针指向目标函数，并且 JVM 也成功初始化后（请阅读 4.3.4 节），Launcher 就会执行 LoadClass()和 GetStaticMethodId()函数。其中 LoadClass()函数用于获取 Java 程序的启动类，而 GetStaticMethodId()函数则用于获取 Java 程序的启动方法。如下所示：

代码 4-7 LoadClass()和 GetStaticMethodId()函数

```
/* 获取 Java 程序的启动类 */
mainClass = LoadClass(env, classname);
/* 获取 Java 程序的启动方法 */
mainID = (*env)->GetStaticMethodID(env, mainClass, "main","
([Ljava/lang/String;)V");
```

当成功获取 Java 程序的启动类和启动方法后，Launcher 就会调用 CallStaticVoidMethod()函数执行 Java 程序的 main()方法，如下所示：

代码 4-8 执行 Java 程序的 main()方法

```
/* 执行 Java 程序的 main()方法 */
(*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs);
```

当 Java 程序执行完后，Launcher 还会通过 JavaVM 类型中的函数指针 DetachCurrentThread 指向本地函数 jni_DetachCurrentThread()断开与主线程之间的连接。如下所示：

代码 4-9 断开与主线程之间的连接

```
/* 断开与主线程之间的连接 */

if ((*vm)->DetachCurrentThread(vm) != 0) {
     message = "Could not detach main thread.";
     messageDest = JNI_TRUE;
     ret = 1;
     goto leave;
}
```

当成功与主线程断开连接后，Launcher 就会一直等待程序中所有的非守护线程（non-daemon）全部执行结束，然后通过 JavaVM 类型中的函数指针 DestroyJavaVM 指向本地函数 jni_DestroyJavaVM()对 JVM 执行销毁。如下所示：

代码 4-10　销毁 JVM

```
/* 销毁 JVM */
(*vm)->DestroyJavaVM(vm);
```

除 jni_DestroyJavaVM()函数可以对 JVM 执行销毁外，在 Java 程序中，开发人员还可以使用 System.exit(0)的方式显式对 JVM 执行销毁。一旦在程序中显式对 JVM 执行销毁后，Launcher 将不会再继续执行 jni_DestroyJavaVM()函数。

为了使大家阅读更方便，本书示例了 JavaMain()函数的完整代码，如下所示：

代码 4-11　JavaMain()函数的完整代码

```
int JNICALL JavaMain(void * _args)
{
    /* 接收 main()函数中传递过来的变量参数 */
    struct JavaMainArgs *args = (struct JavaMainArgs *)_args;
    int argc = args->argc;
    char **argv = args->argv;
    char *jarfile = args->jarfile;
    char *classname = args->classname;
    InvocationFunctions ifn = args->ifn;

    /* 初始化比较重要的一些局部变量 */
    JavaVM *vm = 0;
    JNIEnv *env = 0;
    jstring mainClassName;
    jclass mainClass;
    jmethodID mainID;
    jobjectArray mainArgs;
    int ret = 0;
    jlong start, end;

    /*
     * Error message to print or display; by default the message will
     * only be displayed in a window.
     */
    char * message = "Fatal exception occurred.  Program will exit.";
    jboolean messageDest = JNI_FALSE;

    /* Initialize the virtual machine */

    if (_launcher_debug)
```

```
        start = CounterGet();

    /* 初始化 JVM */
    if (!InitializeJVM(&vm, &env, &ifn)) {
        ReportErrorMessage("Could not create the Java virtual machine.",
                           JNI_TRUE);
        exit(1);
    }

    if (printVersion || showVersion) {
        PrintJavaVersion(env);
        if ((*env)->ExceptionOccurred(env)) {
            ReportExceptionDescription(env);
            goto leave;
        }
        if (printVersion) {
            ret = 0;
            message = NULL;
            goto leave;
        }
        if (showVersion) {
            fprintf(stderr, "\n");
        }
    }

    /* If the user specified neither a class name nor a JAR file */
    if (jarfile == 0 && classname == 0) {
        PrintUsage();
        message = NULL;
        goto leave;
    }

#ifndef GAMMA
    FreeKnownVMs();  /* after last possible PrintUsage() */
#endif

    if (_launcher_debug) {
        end   = CounterGet();
        printf("%ld micro seconds to InitializeJVM\n",
               (long)(jint)Counter2Micros(end-start));
    }

    /* At this stage, argc/argv have the applications' arguments */
    if (_launcher_debug) {
        int i = 0;
        printf("Main-Class is '%s'\n", classname ? classname : "");
        printf("Apps' argc is %d\n", argc);
        for (; i < argc; i++) {
            printf("    argv[%2d] = '%s'\n", i, argv[i]);
```

```
        }
    }

    ret = 1;

    /*
     * Get the application's main class.
     *
     * See bugid 5030265.  The Main-Class name has already been parsed
     * from the manifest, but not parsed properly for UTF-8 support.
     * Hence the code here ignores the value previously extracted and
     * uses the pre-existing code to reextract the value.  This is
     * possibly an end of release cycle expedient.  However, it has
     * also been discovered that passing some character sets through
     * the environment has "strange" behavior on some variants of
     * Windows.  Hence, maybe the manifest parsing code local to the
     * launcher should never be enhanced.
     *
     * Hence, future work should either:
     *     1)   Correct the local parsing code and verify that the
     *          Main-Class attribute gets properly passed through
     *          all environments,
     *     2)   Remove the vestages of maintaining main_class through
     *          the environment (and remove these comments).
     */
    if (jarfile != 0) {
        mainClassName = GetMainClassName(env, jarfile);
        if ((*env)->ExceptionOccurred(env)) {
            ReportExceptionDescription(env);
            goto leave;
        }
        if (mainClassName == NULL) {
          const char * format = "Failed to load Main-Class manifest "
                                "attribute from\n%s";
          message = (char*)JLI_MemAlloc((strlen(format) + strlen(jarfile)) *
                                sizeof(char));
          sprintf(message, format, jarfile);
          messageDest = JNI_TRUE;
          goto leave;
        }
        classname = (char *)(*env)->GetStringUTFChars(env, mainClassName, 0);
        if (classname == NULL) {
            ReportExceptionDescription(env);
            goto leave;
        }

    /* 获取 Java 程序的启动类 */
        mainClass = LoadClass(env, classname);
        if(mainClass == NULL) { /* exception occured */
```

```
            const char * format = "Could not find the main class: %s.
                                  Program will exit.";
            ReportExceptionDescription(env);
            message = (char *)JLI_MemAlloc((strlen(format) +
                                    strlen(classname)) * sizeof(char) );
            messageDest = JNI_TRUE;
            sprintf(message, format, classname);
            goto leave;
        }
        (*env)->ReleaseStringUTFChars(env, mainClassName, classname);
    } else {
      mainClassName = NewPlatformString(env, classname);
      if (mainClassName == NULL) {
        const char * format = "Failed to load Main Class: %s";
        message = (char *)JLI_MemAlloc((strlen(format) + strlen(classname)) *
                                sizeof(char) );
        sprintf(message, format, classname);
        messageDest = JNI_TRUE;
        goto leave;
      }
      classname = (char *)(*env)->GetStringUTFChars(env, mainClassName, 0);
      if (classname == NULL) {
        ReportExceptionDescription(env);
        goto leave;
      }

      /* 获取 Java 程序的启动类 */
      mainClass = LoadClass(env, classname);
      if(mainClass == NULL) { /* exception occured */
        const char * format = "Could not find the main class:
                              %s. Program will exit.";
        ReportExceptionDescription(env);
        message = (char *)JLI_MemAlloc((strlen(format) +
                                    strlen(classname)) * sizeof(char) );
        messageDest = JNI_TRUE;
        sprintf(message, format, classname);
        goto leave;
      }

      (*env)->ReleaseStringUTFChars(env, mainClassName, classname);
    }

    /* 获取 Java 程序的启动方法 */
    mainID = (*env)->GetStaticMethodID(env, mainClass, "main",
                                    "([Ljava/lang/String;)V");
    if (mainID == NULL) {
        if ((*env)->ExceptionOccurred(env)) {
            ReportExceptionDescription(env);
        } else {
```

```
            message = "No main method found in specified class.";
            messageDest = JNI_TRUE;
        }
        goto leave;
    }

    {    /* Make sure the main method is public */
        jint mods;
        jmethodID mid;
        jobject obj = (*env)->ToReflectedMethod(env, mainClass,
                                                mainID, JNI_TRUE);

        if( obj == NULL) { /* exception occurred */
            ReportExceptionDescription(env);
            goto leave;
        }

        mid =
          (*env)->GetMethodID(env,
                              (*env)->GetObjectClass(env, obj),
                              "getModifiers", "()I");
        if ((*env)->ExceptionOccurred(env)) {
            ReportExceptionDescription(env);
            goto leave;
        }

        mods = (*env)->CallIntMethod(env, obj, mid);
        if ((mods & 1) == 0) { /* if (!Modifier.isPublic(mods)) ... */
            message = "Main method not public.";
            messageDest = JNI_TRUE;
            goto leave;
        }
    }

    /* Build argument array */
    mainArgs = NewPlatformStringArray(env, argv, argc);
    if (mainArgs == NULL) {
        ReportExceptionDescription(env);
        goto leave;
    }

    /* 执行 Java 程序的 main()方法 */
    (*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs);

    /*
     * The launcher's exit code (in the absence of calls to
     * System.exit) will be non-zero if main threw an exception.
     */
    ret = (*env)->ExceptionOccurred(env) == NULL ? 0 : 1;
```

```
    /* 断开与主线程之间的连接 */
    if ((*vm)->DetachCurrentThread(vm) != 0) {
        message = "Could not detach main thread.";
        messageDest = JNI_TRUE;
        ret = 1;
        goto leave;
    }

    message = NULL;

 leave:
    /* 销毁 JVM */
    (*vm)->DestroyJavaVM(vm);

    if(message != NULL && !noExitErrorMessage)
      ReportErrorMessage(message, messageDest);
    return ret;
}
```

4.3.4 调用JNI_CreateJavaVM()函数初始化HotSpot

本书 4.3.3 节介绍了结构体 InvocationFunctions 类型中包含的函数指针 CreateJavaVM 类型，那么与 CreateJavaVM 类型对应的目标函数就是 JNI_CreateJavaVM()。

当在 Launcher 中通过 JavaMain()函数调用 InitializeJVM()函数时，InitializeJVM()函数便会通过函数指针 CreateJavaVM 类型指向本地函数 JNI_CreateJavaVM()完成 HotSpot 的初始化。

为了使大家阅读更方便，本书示例了 InitializeJVM()函数的完整代码，如下所示：

代码 4-12 InitializeJVM()函数的完整代码

```
static jboolean InitializeJVM
  (JavaVM **pvm, JNIEnv **penv, InvocationFunctions *ifn){
    JavaVMInitArgs args;
    jint r;

    memset(&args, 0, sizeof(args));
    args.version  = JNI_VERSION_1_2;
    args.nOptions = numOptions;
    args.options  = options;
    args.ignoreUnrecognized = JNI_FALSE;

    if (_launcher_debug) {
        int i = 0;
        printf("JavaVM args:\n    ");
        printf("version 0x%08lx, ", (long)args.version);
```

```
        printf("ignoreUnrecognized is %s, ",
               args.ignoreUnrecognized ? "JNI_TRUE" : "JNI_FALSE");
        printf("nOptions is %ld\n", (long)args.nOptions);
        for (i = 0; i < numOptions; i++)
            printf("    option[%2d] = '%s'\n",
                   i, args.options[i].optionString);
    }

    /* 通过函数指针指向本地的 JNI_CreateJavaVM()函数完成 JVM 初始化 */
    r = ifn->CreateJavaVM(pvm, (void **)penv, &args);
    JLI_MemFree(options);
    return r == JNI_OK;
}
```

在本地函数 JNI_CreateJavaVM()内部，究竟是如何执行 JVM 的初始化呢？大家可以在 JNI 接口中找到 HotSpot 的具体初始化细节（请阅读第 5 章）。简单来说，JNI_CreateJavaVM()函数又调用了包含在/hotspot/src/share/vm/runtim/thread.cpp 中 Threads 模块的 create_vm()函数来最终完成 HotSpot 的初始化。如下所示：

代码 4-13　调用 Threads 模块的 create_vm()函数

```
/* 初始化 JVM */
result = Threads::create_vm((JavaVMInitArgs*) args, &can_try_again);
```

4.3.5　调用LoadClass()函数获取Java启动类

当成功对 JVM 执行初始化后，Launcher 就会调用 LoadClass()函数获取 Java 程序的启动类。如下所示：

代码 4-14　获取 Java 启动类

```
/* 获取 Java 程序的启动类 */
mainClass = LoadClass(env, classname);
```

之所以获取 Java 程序的启动类，是因为后续在调用 CallStaticVoidMethod()函数执行 Java 程序的 main()方法之前，需要将其作为参数进行传递。并且在 LoadClass()函数内部，FindClass()函数会通过所传入的启动类名称装载目标类型。

为了使大家阅读更方便，本书示例了 LoadClass()函数的完整代码，如下所示：

代码 4-15　LoadClass()函数的完整代码

```
static jclass LoadClass(JNIEnv *env, char *name)
{
    char *buf = JLI_MemAlloc(strlen(name) + 1);
```

```
    char *s = buf, *t = name, c;
    jclass cls;
    jlong start, end;

    if (_launcher_debug)
        start = CounterGet();

    do {
        c = *t++;
        *s++ = (c == '.') ? '/' : c;
    } while (c != '\0');

    /* 装载目标类型 */
    cls = (*env)->FindClass(env, buf);
    JLI_MemFree(buf);

    if (_launcher_debug) {
        end   = CounterGet();
        printf("%ld micro seconds to load main class\n",
               (long)(jint)Counter2Micros(end-start));
        printf("----_JAVA_LAUNCHER_DEBUG----\n");
    }

    return cls;
}
```

4.3.6 调用GetStaticMethodId()函数获取Java启动方法

当 Launcher 成功获取到 Java 程序的启动类后，就会调用 GetStaticMethodId()函数获取 Java 程序的启动方法。其实 GetStaticMethodId 属于结构体 JNIEnv 类型的函数指针，该类型对应的目标函数为 jni_GetStaticMethodId()。如下所示：

代码 4-16 获取 Java 程序的启动方法

```
/* 获取 Java 程序的启动方法 */
mainID = (*env)->GetStaticMethodID(env, mainClass, "main",
  "([Ljava/lang/String;)V");
```

和 LoadClass()函数一样，在调用 CallStaticVoidMethod()函数执行 Java 程序的 main()方法之前，同样需要将其作为参数进行传递。

4.3.7 调用CallStaticVoidMethod()函数执行Java启动方法

当成功获取 Java 程序的启动类和启动方法后，Launcher 就会调用 CallStaticVoidMethod()

函数执行 Java 程序的 main()方法，如下所示：

代码 4-17　执行 Java 程序的 main()方法

```
/* 执行 Java 程序的 main()方法 */
(*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs);
```

CallStaticVoidMethod()函数是属于结构体 JNIEnv 类型的函数指针，该类型对应的目标函数为 jni_CallStaticVoidMethod()。而在 jni_CallStaticVoidMethod()函数内部又调用了包含在/hotspot/src/share/vm/prims/jni.cpp 中的 jni_invoke_static()函数来执行 Java 程序的 main()方法。如下所示：

代码 4-18　jni_invoke_static()函数的完整代码

```
static void jni_invoke_static(JNIEnv *env, JavaValue* result,
  jobject receiver, JNICallType call_type, jmethodID method_id,
  JNI_ArgumentPusher *args, TRAPS) {
  methodHandle method(THREAD, JNIHandles::resolve_jmethod_id(method_id));

  // Create object to hold arguments for the JavaCall, and associate
  // it with the jni parser
  ResourceMark rm(THREAD);
  int number_of_parameters = method->size_of_parameters();
  JavaCallArguments java_args(number_of_parameters);
  args->set_java_argument_object(&java_args);

  assert(method->is_static(), "method should be static");

  // Fill out JavaCallArguments object
  args->iterate( Fingerprinter(method).fingerprint() );
  // Initialize result type
  result->set_type(args->get_ret_type());

  // 通过 JavaCalls 类型调用 Java 程序的 main()方法
  JavaCalls::call(result, method, &java_args, CHECK);

  // Convert result
  if (result->get_type() == T_OBJECT || result->get_type() == T_ARRAY) {
    result->set_jobject
      (JNIHandles::make_local(env, (oop) result->get_jobject()));
  }
}
```

在 jni_invoke_static()函数内部，JavaCalls 首先会从 GetStaticMethodId()函数中获取出一个全局唯一的方法 ID，然后将其转换为方法句柄后，再通过 call()函数最终执行 Java 程序的 main()方法。除可以执行 Java 程序的 main()方法外，程序中所有的 Java 方法执行都需

要依赖 JavaCalls 模块完成调用，JavaCalls 的头文件含在/hotspot/src/share/vm/runtime/javaCalls.hpp 中。

为了使大家阅读更方便，本书示例了 JavaCalls 模块中声明的所有调用接口，如下所示：

代码 4-19 JavaCalls 模块中声明的调用接口

```
class JavaCalls: AllStatic {
  static void call_helper(JavaValue* result, methodHandle* method,
    JavaCallArguments* args, TRAPS);
  public:
  // Optimized Constuctor call
  static void call_default_constructor(JavaThread* thread,
    methodHandle method, Handle  receiver, TRAPS);

  // call_special
  // The receiver must be first oop in argument list
  static void call_special(JavaValue* result, KlassHandle klass,
  Symbol* name, Symbol* signature, JavaCallArguments* args, TRAPS);

  static void call_special(JavaValue* result, Handle receiver,
    KlassHandle klass, Symbol* name, Symbol* signature, TRAPS); // No args
  static void call_special(JavaValue* result, Handle receiver,
    KlassHandle klass, Symbol* name, Symbol* signature, Handle arg1,
    TRAPS);
  static void call_special(JavaValue* result, Handle receiver,
    KlassHandle klass, Symbol* name,Symbol* signature, Handle arg1,
    Handle arg2, TRAPS);

  // virtual call
  // The receiver must be first oop in argument list
  static void call_virtual(JavaValue* result, KlassHandle spec_klass,
  Symbol* name, Symbol* signature, JavaCallArguments* args, TRAPS);

  static void call_virtual(JavaValue* result, Handle receiver,
    KlassHandle spec_klass, Symbol*name, Symbol* signature, TRAPS);
    // No args
  static void call_virtual(JavaValue* result, Handle receiver,
    KlassHandle spec_klass, Symbol*name, Symbol* signature, Handle arg1,
    TRAPS);
  static void call_virtual(JavaValue* result, Handle receiver,
    KlassHandle spec_klass, Symbol*name, Symbol* signature, Handle arg1,
    Handle arg2, TRAPS);

  // Static call
  static void call_static(JavaValue* result, KlassHandle klass,
    Symbol* name, Symbol* signature,JavaCallArguments* args, TRAPS);

  static void call_static(JavaValue* result, KlassHandle klass,
```

```
    Symbol* name, Symbol* signature,TRAPS);
  static void call_static(JavaValue* result, KlassHandle klass,
    Symbol* name, Symbol* signature,Handle arg1, TRAPS);
  static void call_static(JavaValue* result, KlassHandle klass,
    Symbol* name, Symbol* signature, Handle arg1, Handle arg2, TRAPS);

  // Low-level interface
  static void call(JavaValue* result, methodHandle method,
    JavaCallArguments* args, TRAPS);
};
```

4.3.8 调用jni_DestroyJavaVM函数销毁HotSpot

当成功与主线程断开连接后，Launcher 就会一直等待程序中所有的非守护线程（non-daemon）全部执行结束，然后通过 JavaVM 类型中的函数指针 DestroyJavaVM 指向本地函数 jni_DestroyJavaVM()对 HotSpot 执行销毁。

而包含在/hotspot/src/share/vm/prims/jni.cpp 中的 jni_DestroyJavaVM()函数内部又调用了 Threads 模块的 destroy_vm()函数最终完成 HotSpot 的销毁。如下所示：

代码 4-20 最终销毁 JVM

```
jint JNICALL jni_DestroyJavaVM(JavaVM *vm) {
#ifndef USDT2
  DTRACE_PROBE1(hotspot_jni, DestroyJavaVM__entry, vm);
#else /* USDT2 */
  HOTSPOT_JNI_DESTROYJAVAVM_ENTRY(
                                vm);
#endif /* USDT2 */
  jint res = JNI_ERR;
  DT_RETURN_MARK(DestroyJavaVM, jint, (const jint&)res);

  if (!vm_created) {
    res = JNI_ERR;
    return res;
  }

  JNIWrapper("DestroyJavaVM");
  JNIEnv *env;
  JavaVMAttachArgs destroyargs;
  destroyargs.version = CurrentVersion;
  destroyargs.name = (char *)"DestroyJavaVM";
  destroyargs.group = NULL;
  res = vm->AttachCurrentThread((void **)&env, (void *)&destroyargs);
  if (res != JNI_OK) {
    return res;
```

```
    }

    // Since this is not a JVM_ENTRY we have to set the thread state manually
         before entering.
    JavaThread* thread = JavaThread::current();
    ThreadStateTransition::transition_from_native(thread, _thread_in_vm);

   //启动线程销毁 JVM
   if (Threads::destroy_vm()) {
      // Should not change thread state, VM is gone
      vm_created = false;
      res = JNI_OK;
      return res;
    } else {
      ThreadStateTransition::transition_and_fence(thread, _thread_in_vm,
         _thread_in_native);
      res = JNI_ERR;
      return res;
    }
  }
```

4.4 实战：在Launcher中添加自定义函数模块

当大家理解 Launcher 的执行过程后，便可以开始着手尝试在 Launcher 的源码中添加一些自定义的函数模块。因为这样一来，当你在实际的开发过程中需要通过修改 HotSpot 的源码满足特定的业务场景时，自然会有所帮助，毕竟勇于迈出第一步是非常重要的。

在本章的实战小节中，我们要做的仅仅只是在 Launcher 调用 CallStaticVoidMethod()函数执行之前，在自定义函数中引用传递给 CallStaticVoidMethod()函数的一些变量参数。如下所示：

代码 4-21 添加 setParameters()函数

```
/* 在调用 CallStaticVoidMethod()函数之前，引用其变量参数 */
setParameters(env, mainClass, mainID, mainArgs);

/* 执行 Java 程序的 main()方法 */
(*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs);
```

上述语句中，笔者在 CallStaticVoidMethod()函数执行之前调用了自定义函数 setParameters()引用了传递给 CallStaticVoidMethod()函数的一些变量参数。其实 setParameters()函数中所包含的功能只是引用了这些传递过来的变量参数，并输出一串字符信息而已。如下所示：

代码 4-22 自定义 setParameters()函数

```
/*
* 引用变量参数信息
*/
void getParameters(jstring mainClassName,JNIEnv *env,jclass mainClass,
     jmethodID mainID, jobjectArray mainArgs){
          printf("成功接收到变量参数...");
}
```

当成功将 Launcher 的源码重新编译后，大家便可以使用 GDB 工具调试上述这段代码。当程序执行到 setParameters()函数时，输入命令“step”将会进入到该函数的内部；再次输入命令“info args”后，便会输出 setParameters()函数引用的这些变量的参数值，如下所示：

```
(gdb) info args
mainClassName = 0xb680a5dc
env = 0xb6809148
mainClass = 0xb680a5e0
mainID = 0xb6858990
mainArgs = 0xb680a5f0
```

本书 1.5.2 节中介绍了基于 OpenJDK 深度定制的 TaobaoVM，在此笔者不得不佩服和认同淘宝的技术实力。淘宝的技术团队敢于做国内第一个吃螃蟹的人，不仅深入到 HotSpot 的源码底层修改大量的实现细节，更是在原有基础之上扩充了适应自身业务特点的一些新特性。单从这两点来看，淘宝做得确实非常不错。如果大家同样对此感兴趣，也可以尝试着往更深层次的方面修改 HotSpot 的源码或者扩充新的功能实现，这样不仅可以帮助你更好地理解 HotSpot，还能让你在 JVM 领域拥有更深的造诣。

4.5 本章小结

本章笔者深入源码剖析了 Launcher 的实现细节和执行过程，其中涵盖了 Launcher 中用于创建运行环境的启动函数 main()、主线程函数 JavaMain()、初始化 JVM 的 InitializeJVM()函数、获取 Java 程序启动类的 LoadClass()函数、获取 Java 启动方法的 getStaticMethodId()函数、调用 Java 程序 main()方法的 CallStaticVoidMethod()函数、断开主线程连接的 DetachCurrentThread()函数和销毁 JVM 的 DestroyJavaVM()函数。

当大家理解了本章每节所阐述的内容后，再亲自动手使用 GDB 或者其他 IDE 工具进行调试跟踪，相信这会对你在后续章节的学习中有所帮助。

第 5 章

剖析HotSpot的初始化过程

之前笔者在第 4 章中曾经提及过，由 Launcher 负责调用 HotSpot 的核心代码对 JVM 执行初始化，以及由它负责维护 JVM 的整个生命周期。那么 JVM 的初始化操作其实就是 HotSpot 执行启动的前提条件，并且在初始化过程中还涉及 HotSpot 中一些核心模块的初始化（比如初始化 os 模块、初始化全局数据结构、启动线程、初始化全局模块等），所以了解 HotSpot 的初始化过程对于理解其整体架构具有非常重要的意义。本章接下来将会围绕 HotSpot 中的部分构成模块及其初始化过程进行讲解。

5.1 HotSpot的构成模块

在前面几个章节中，本书详细地讲解了关于 HotSpot 源码的部分内容，其中涵盖了 HotSpot 的源码下载、编译以及使用 GDB 工具进行 Debug；在上一章，本书又深入到了 Launcher 的源码细节中，为大家剖析了 Launcher 是如何启动 HotSpot 的。而本章则会延续之前的内容，为大家深入剖析 HotSpot 的初始化过程。

我们都知道是由 Launcher 负责维护 JVM 的整个生命周期，那么当 Launcher 中的启动函数 main()创建主线程并调用 JavaMain()函数时，会由 InvocationFunctions 类型的函数指针 CreateJavaJVM 指向本地函数 JNI_CreateJavaJVM()完成 JVM 的初始化工作，而在 JNI_CreateJavaJVM()函数内部却又调用了 Threads 模块的 create_vm()函数来最终完成 JVM 的初始化。在正式开始讲解 create_vm()函数究竟是如何初始化 JVM 之前，大家首先回顾一下，在 Launcher 启动 HotSpot 时，大部分的功能是否都是通过函数指针指向本地函数，再由本地函数调用指定模块的具体函数去实现的呢？那么由此可见，这些功能模块就是构成 HotSpot 整体结构的核心，如图 5-1 所示。

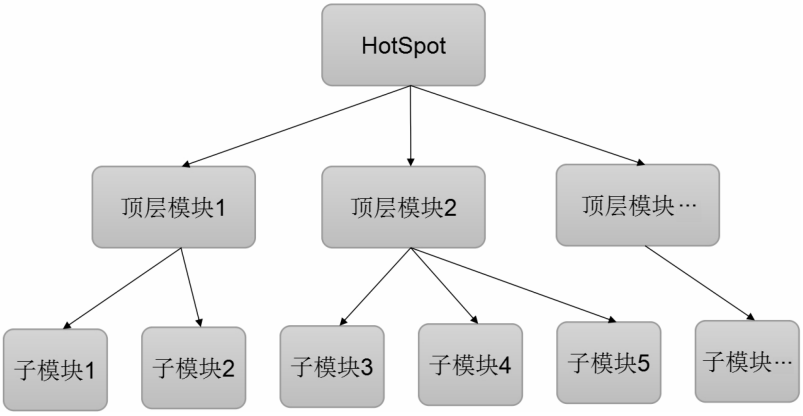


图 5-1 HotSpot 的构成模块

为了让大家能够更好地理解 HotSpot 中每一个模块所负责的功能，以及提供一个思路，让大家能够更清楚地知道具体的功能究竟是包含在哪一个模块下，所以接下来笔者将带领大家一同去认识包含在 HotSpot 中的这些功能模块。HotSpot 的顶层模块包含在 /openjdk/hotspot/src/share/vm 中，分别为 adlc、asm、c1、ci、classfile、code、compiler、gc、interpreter、libadt、memory、oops、opto、Prims、runtime、services、shark 和 utilities。

尽管 HotSpot 中的顶层模块比较多，但理解每一个模块所包含的主要功能其实并不困难。其中：

- adlc 模块中所包含的功能为平台描述文件的编译器。
- asm 模块中所包含的功能为汇编器接口。
- c1 模块中所包含的功能为 client 编译器。
- ci 模块中所包含的功能为动态编译器的公共服务/从动态编译器到 VM 的接口。
- classFile 模块中所包含的功能为类文件的处理（包括类加载和系统符号表等）。
- code 模块中所包含的功能为动态生成的代码的管理。
- compiler 模块中所包含的功能为从 VM 调用动态编译器的接口。
- gc 模块由 gc_interface 和 gc_implementation 两部分构成，其中 gc_interface 模块中所包含的功能为 GC 的接口，而 gc_implementation 模块中所包含的功能为 GC 的实现；并且该模块中还包含了 concurrentMarkSweep、g1、parallelScavenge、parNew 和 shared 这 5 个重要的子模块，concurrentMarkSweep 子模块中所包含的功能为 Concurrent Mark Sweep GC 的实现，g1 子模块中所包含的功能为 Garbage-First GC 的实现（不使用老的分代式 GC 框架），parallelScavenge 子模块中所包含的功能为 ParallelScavenge GC 的实现（server VM 默认不使用老的分代式 GC 框架），parNew 子模块中所包含的功能为 ParNew GC 的实现，shared 子模块中所包含的功能为 GC

的共通实现。

- interpreter 模块中所包含的功能为内部解释器，包括“模板解释器”（官方版在用）和“C++解释器”（官方版不在用）。
- libadt 模块中所包含的功能为一些抽象数据结构。
- memory 模块中所包含的功能为内存管理相关（老的分代式 GC 框架也在这里）。
- oops 模块中所包含的功能为 HotSpot VM 的对象系统的实现。
- opto 模块中所包含的功能为 server 编译器（又称“C2”或“Opto”）。
- Prims 模块中所包含的功能为 HotSpot VM 的对外接口，包括部分标准库的 native 部分和 JVMTI 实现。
- runtime 模块中所包含的功能为运行时支持库（包括线程管理、编译器调度、锁、反射等）。
- services 模块中所包含的功能主要是用来支持 JMX 之类的管理功能的接口。
- shark 模块中所包含的功能为基于 LLVM 的 JIT 编译器（官方版不在用）。
- utilities 模块中所包含的功能则为一些基本的工具类。

5.2 Prims模块

在 HotSpot 中，Prims 模块的主要任务就是负责提供外部程序访问 JVM 内部信息的对外访问接口，这些接口主要包括了 JNI、JVM、JVMTI 和 Perf，如图 5-2 所示。

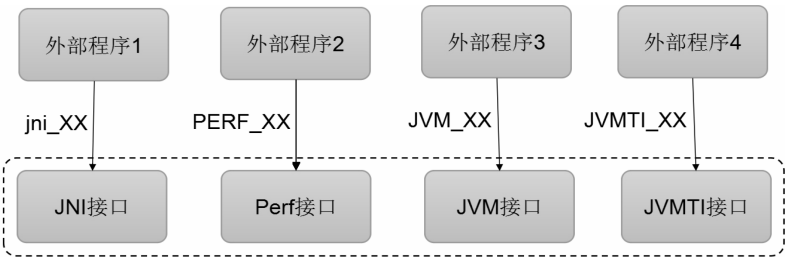


图 5-2 Prims 模块的对外访问接口

5.2.1 JNI子模块

JNI（Java Native Interface，Java 本地接口）技术早在 Java1.1 版本的时候就已经归纳为 Java 平台标准的一部分，那么究竟在什么情况下需要用到 JNI 呢？我们都知道 Java 源码的编译结果为字节码，字节码并非是本地机器指令，而是一种符合 Java 虚拟机规范并能够被其所识别及运行的高度优化指令。也就是说，Java 源码的编译结果屏蔽了与底层操作系统和物理硬件相关的一些特性，从跨平台的角度来看，开发人员只需关注于自身业务即可。

但是如果希望使用 Java 程序去访问底层操作系统和物理硬件资源时则显得无能为力，那么这个时候就需要使用 JNI 技术让 Java 代码和使用其他编程语言编写的代码进行交互。在 HotSpot 的源码中定义了许多以“jni_”开头的函数，这些函数都是由 C/C++代码编写的本地代码，用于让外部程序调用 JVM 中的本地库文件。

当然在程序中使用 JNI 技术调用本地方法其实是非常简单的。首先只需在 Java 代码中声明一个使用关键字 native 修饰的抽象方法，然后将编写好的 Java 代码编译为字节码文件后使用命令“javah -jni”即可生成一个对应的 jni 头文件，最后编写 C/C++代码并生成动态链接库后再由之前 Java 代码中的 static 代码块动态加载即可成功调用本地代码。如图 5-3 所示。

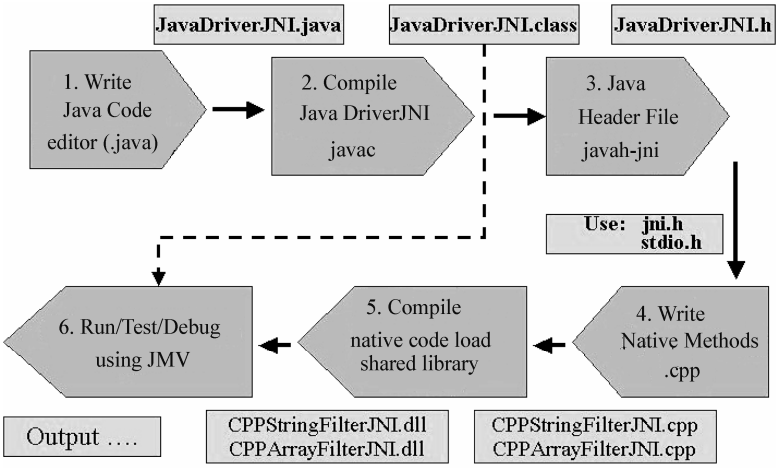


图 5-3 使用 JNI 技术调用本地方法

5.2.2 JVM子模块

JVM 模块与 JNI 模块其实息息相关，这是因为 JVM 模块在 JNI 模块的功能之上进行了扩充；但和 JNI 模块不同的是，JVM 模块中所包含的本地方法全都是与 JVM 相关的底层函数。也就是说，以“JVM_”开头的函数全都是由本地代码实现的 Java API。

大家经常使用到的 Java API 中的功能，其实大部分都需要调用本地代码去实现，这些功能分别包括：java.lang.Object、java.lang.String、java.lang.System、Thread 操作、I/O 操作等。对于开发人员而言，或许并不关心这些底层实现，但实际上 Java 程序自身是无法实现这些功能的。为了使大家阅读更方便，本书示例了包含在/openjdk/hotspot/src/share/vm/Prims/jvm.h 中的部分由本地代码实现的 Java API。如下所示：

代码 5-1 jvm.h 的代码片段

```
/*
 * java.lang.Object
 */
JNIEXPORT jint JNICALL
JVM_IHashCode(JNIEnv *env, jobject obj);

JNIEXPORT void JNICALL
JVM_MonitorWait(JNIEnv *env, jobject obj, jlong ms);

JNIEXPORT void JNICALL
JVM_MonitorNotify(JNIEnv *env, jobject obj);

JNIEXPORT void JNICALL
JVM_MonitorNotifyAll(JNIEnv *env, jobject obj);

JNIEXPORT jobject JNICALL
JVM_Clone(JNIEnv *env, jobject obj);

/*
 * java.lang.String
 */
JNIEXPORT jstring JNICALL
JVM_InternString(JNIEnv *env, jstring str);

/*
 * java.lang.System
 */
JNIEXPORT jlong JNICALL
JVM_CurrentTimeMillis(JNIEnv *env, jclass ignored);

JNIEXPORT jlong JNICALL
JVM_NanoTime(JNIEnv *env, jclass ignored);

JNIEXPORT void JNICALL
JVM_ArrayCopy(JNIEnv *env, jclass ignored, jobject src, jint src_pos,
              jobject dst, jint dst_pos, jint length);

JNIEXPORT jobject JNICALL
JVM_InitProperties(JNIEnv *env, jobject p);

/*
 * java.io.File
 */
JNIEXPORT void JNICALL
JVM_OnExit(void (*func)(void));

/*
```

```
 * java.lang.Thread
 */
JNIEXPORT void JNICALL
JVM_StartThread(JNIEnv *env, jobject thread);

JNIEXPORT void JNICALL
JVM_StopThread(JNIEnv *env, jobject thread, jobject exception);

JNIEXPORT jboolean JNICALL
JVM_IsThreadAlive(JNIEnv *env, jobject thread);

JNIEXPORT void JNICALL
JVM_SuspendThread(JNIEnv *env, jobject thread);

JNIEXPORT void JNICALL
JVM_ResumeThread(JNIEnv *env, jobject thread);

JNIEXPORT void JNICALL
JVM_SetThreadPriority(JNIEnv *env, jobject thread, jint prio);

JNIEXPORT void JNICALL
JVM_Yield(JNIEnv *env, jclass threadClass);

JNIEXPORT void JNICALL
JVM_Sleep(JNIEnv *env, jclass threadClass, jlong millis);

JNIEXPORT jobject JNICALL
JVM_CurrentThread(JNIEnv *env, jclass threadClass);

JNIEXPORT jint JNICALL
JVM_CountStackFrames(JNIEnv *env, jobject thread);

JNIEXPORT void JNICALL
JVM_Interrupt(JNIEnv *env, jobject thread);

JNIEXPORT jboolean JNICALL
JVM_IsInterrupted(JNIEnv *env, jobject thread, jboolean clearInterrupted);

JNIEXPORT jboolean JNICALL
JVM_HoldsLock(JNIEnv *env, jclass threadClass, jobject obj);

JNIEXPORT void JNICALL
JVM_DumpAllStacks(JNIEnv *env, jclass unused);

JNIEXPORT jobjectArray JNICALL
JVM_GetAllThreads(JNIEnv *env, jclass dummy);

JNIEXPORT void JNICALL
JVM_SetNativeThreadName(JNIEnv *env, jobject jthread, jstring name);
```

```
/* getStackTrace() and getAllStackTraces() method */
JNIEXPORT jobjectArray JNICALL
JVM_DumpThreads(JNIEnv *env, jclass threadClass, jobjectArray threads);

/*
 * java.io.ObjectInputStream
 */
JNIEXPORT jobject JNICALL
JVM_AllocateNewObject(JNIEnv *env, jobject obj, jclass currClass,
                      jclass initClass);

JNIEXPORT jobject JNICALL
JVM_AllocateNewArray(JNIEnv *env, jobject obj, jclass currClass,
                     jint length);

JNIEXPORT jobject JNICALL
JVM_LatestUserDefinedLoader(JNIEnv *env);
```

5.2.3 JVMTI子模块

JVMTI（Java Virtual Machine Tool Interface，Java 虚拟机工具接口）是在 Java5 版本的时候提供的一种用于取代 JVMPI（Java Virtual Machine Profiling Interface，Java 虚拟机剖析接口）和 JVMDI（Java Virtual Machine Debugging Interface，Java 虚拟机调试接口）的编程接口。开发人员可以通过使用 JVMTI 接口创建代理程序（Agent）对 Java 程序进行监视和控制，包括内存使用率、CPU 使用率和线程分析等。JVMTI 的架构模型，如图 5-4 所示。

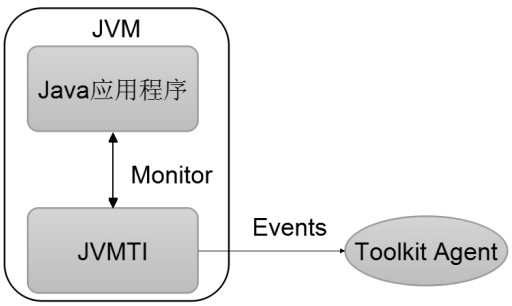


图 5-4 JVMTI 架构模型

Agent 可以向运行中的 JVM 实例订阅感兴趣的事件，当这些事件发生时，会以事件回调函数的方式激活代理程序；同时 JVMTI 提供了众多的功能函数，以查询和控制 Java 程序的运行状态。Agent 通过 JVMTI 所提供的接口与 JVM 进行通信，并同步监控 JVM 的运行状态，它与运行中的 Java 程序是彼此独立的，不会影响到程序的正常运行。Agent 可以使用任何支持 C 语言标准的本地语言进行编写，并以动态链接库的方式存在，Java 程序启动的

时候加载这个动态链接库即可。基于 JVMTI 接口构建的 Agent 可以方便地实现对 Java 线程状态切换的跟踪，从而使得开发人员能够在程序运行时清楚地了解线程的工作情况，方便分析和调试。

5.2.4 Perf子模块

除了 JNI 模块、JVM 模块和 JVMTI 模块，在 Prims 模块中还包含一个子模块，那就是 Pref 模块。该模块是 sun.misc.Perf 类的底层实现，函数名称都以“PERF_”开头，用于监控 JVM 内部的 Perf Data 计数器。

5.3 Runtime模块

由于 HotSpot 中所包含的顶层模块和子模块非常多，所以本章仅会对 Prim 和 Runtime 这两个与 JVM 初始化密切相关的模块进行讲解，而关于其他模块，大家则可以通过阅读源码或者查阅其他 JVM 文档的方式进行了解和学习，本书则不再过多阐述。

Runtime 模块在 HotSpot 中扮演着非常重要的角色。顾名思义，Runtime 其实就是运行时模块，该模块内部包含了许多子模块，比如定义了各类线程的 Thread 子模块、标记和传递 JVM 参数和选项的 Arguments 子模块、生成 stub 的 StubRoutines 和 StubCodeGenerator 子模块、表示物理栈帧的 Frame 子模块、匹配编译策略的 CompilationPolicy 子模块、os 初始化的 Init 子模块、用于创建其他线程的 VMThread 子模块和虚拟机核心操作的 VMOperation 子模块等。为了使大家阅读更方便，本书示例了包含在/openjdk/hotspot/src/share/vm/runtime/vm_operations.hpp 中的 VMOperation 模块定义的核心操作模板。如下所示：

代码 5-2 VMOperation 模块定义的核心操作模板

```
#define VM_OPS_DO(template)
template(Dummy)
template(ThreadStop)
template(ThreadDump)
template(PrintThreads)
template(FindDeadlocks)
template(ForceSafepoint)
template(ForceAsyncSafepoint)
template(Deoptimize)
template(DeoptimizeFrame)
template(DeoptimizeAll)
template(ZombieAll)
template(UnlinkSymbols)
template(HandleFullCodeCache)
template(Verify)
```

```
template(PrintJNI)
template(HeapDumper)
template(DeoptimizeTheWorld)
template(GC_HeapInspection)
template(GenCollectFull)
template(GenCollectFullConcurrent)
template(GenCollectForAllocation)
template(GenCollectForPermanentAllocation)
template(ParallelGCFailedAllocation)
template(ParallelGCFailedPermanentAllocation)
template(ParallelGCSystemGC)
template(CGC_Operation)
template(CMS_Initial_Mark)
template(CMS_Final_Remark)
template(G1CollectFull)
template(G1CollectForAllocation)
template(G1IncCollectionPause)
template(EnableBiasedLocking)
template(RevokeBias)
template(BulkRevokeBias)
template(PopulateDumpSharedSpace)
template(JNIFunctionTableCopier)
template(RedefineClasses)
template(GetOwnedMonitorInfo)
template(GetObjectMonitorUsage)
template(GetCurrentContendedMonitor)
template(GetStackTrace)
template(GetMultipleStackTraces)
template(GetAllStackTraces)
template(GetThreadListStackTraces)
template(GetFrameCount)
template(GetFrameLocation)
template(ChangeBreakpoints)
template(GetOrSetLocal)
template(GetCurrentLocation)
template(EnterInterpOnlyMode)
template(ChangeSingleStep)
template(HeapWalkOperation)
template(HeapIterateOperation)
template(ReportJavaOutOfMemory)
template(JFRCheckpoint)
template(Exit)
template(LinuxDllLoad)
```

模板（Template）是 C++编程中的利器，可是对于 Java 开发人员而言，对于模板的概念或许会稍微感觉到有些陌生，但是 C++的模板却和 Java 的泛型语法息息相关。简单来说，Java 泛型语法的设计最早就是起源于 C++中的模板，只不过 Java 泛型从某种意义上来说并

非是真正意义上的泛型操作，而是“伪泛型”。Java 泛型代码曾经受到一个最主要的制约就是 JVM 需要同时兼容泛型代码与非泛型代码，因此 Java 的设计者们引入了擦除特性（erasure）来对泛型代码进行约束。所谓擦除特性指的就是 Java 源码中的泛型信息只允许停留在编译前期，而编译后的字节码文件中将不再保留任何的泛型信息。也就是说，泛型信息在编译时将会被全部删除，其中泛型类型的类型参数则会被替换为 Object 类型，并在实际使用时强制转换为指定的目标数据类型，而 C++中的模板则会在编译时就将模板类型中的类型参数根据所传递的指定数据类型生成相对应的目标代码。

5.3.1 Thread子模块

Thread 子模块中定义了 JVM 中的两种线程类型，分别是守护线程（daemon thread）和非守护线程（non-daemon thread）。其次该模块中还包含一个 Threads 子模块用于维护 JVM 中的线程队列，并由 Threads 模块中的 create_vm()函数最终完成 JVM 的初始化。

5.3.2 调用create_vm()函数完成HotSpot的最终初始化

HotSpot 的初始化过程比较复杂，步骤也比较繁多，其中比较重要的几个初始化步骤，分别是 os 模块的初始化、启动线程、全局数据结构的初始化、全局模块的初始化等。为了使大家阅读更方便，本书示例了包含在/openjdk/hotspot/src/share/vm/runtime/thread.cpp 中的 create_vm()函数的完整代码。如下所示：

代码 5-3 create_vm()函数的完整代码

```
jint Threads::create_vm(JavaVMInitArgs* args, bool* canTryAgain) {
  extern void JDK_Version_init();

  /* 检查版本 */
  if (!is_supported_jni_version(args->version)) return JNI_EVERSION;

  /* 初始化输出流模块 */
  ostream_init();

  /* 配置 Launcher 的一些相关参数 */
  Arguments::process_sun_java_launcher_properties(args);

  /* 在使用 TLS 之前初始化 os 模块 */
  os::init();

  /* 初始化系统属性 */
  Arguments::init_system_properties();

  // So that JDK version can be used as a discriminator when parsing arguments
```

```
    JDK_Version_init();

    // Update/Initialize System properties after JDK version number is known
    Arguments::init_version_specific_system_properties();

    /* 参数解析 */
    jint parse_result = Arguments::parse(args);
    if (parse_result != JNI_OK) return parse_result;

    if (PauseAtStartup) {
      os::pause();
    }

  #ifndef USDT2
    HS_DTRACE_PROBE(hotspot, vm__init__begin);
  #else /* USDT2 */
    HOTSPOT_VM_INIT_BEGIN();
  #endif /* USDT2 */

    /* 记录虚拟机的创建时间 */
    TraceVmCreationTime create_vm_timer;
    create_vm_timer.start();

    // Timing (must come after argument parsing)
    TraceTime timer("Create VM", TraceStartupTime);

    /* 解析 ARGS 后对内存、栈、线程等与 os 模块密切相关的部分进行初始化 */
    jint os_init_2_result = os::init_2();
    if (os_init_2_result != JNI_OK) return os_init_2_result;

    /* 初始化 TLS */
    ThreadLocalStorage::init();

    // Bootstrap native memory tracking, so it can start recording memory
    // activities before worker thread is started. This is the first phase
    // of bootstrapping, VM is currently running in single-thread mode.
    MemTracker::bootstrap_single_thread();

    /* 初始化输出流记录 */
    ostream_init_log();

    // Convert -Xrun to -agentlib: if there is no JVM_OnLoad
    // Must be before create_vm_init_agents()
    if (Arguments::init_libraries_at_startup()) {
      convert_vm_init_libraries_to_agents();
    }

    // Launch -agentlib/-agentpath and converted -Xrun agents
    if (Arguments::init_agents_at_startup()) {
```

```
  create_vm_init_agents();
}

// Initialize Threads state
_thread_list = NULL;
_number_of_threads = 0;
_number_of_non_daemon_threads = 0;

/* 初始化全局数据结构，并在堆上创建系统类 */
vm_init_globals();

// Attach the main thread to this os thread
JavaThread* main_thread = new JavaThread();
main_thread->set_thread_state(_thread_in_vm);
// must do this before set_active_handles and initialize_thread_local_storage
// Note: on solaris initialize_thread_local_storage() will (indirectly)
// change the stack size recorded here to one based on the java thread
// stacksize. This adjusted size is what is used to figure the placement
// of the guard pages.
main_thread->record_stack_base_and_size();
main_thread->initialize_thread_local_storage();

main_thread->set_active_handles(JNIHandleBlock::allocate_block());

if (!main_thread->set_as_starting_thread()) {
  vm_shutdown_during_initialization(
    "Failed necessary internal allocation. Out of swap space");
  delete main_thread;
  *canTryAgain = false; // don't let caller call JNI_CreateJavaVM again
  return JNI_ENOMEM;
}

// Enable guard page *after* os::create_main_thread(), otherwise it
// would crash Linux VM, see notes in os_linux.cpp.
main_thread->create_stack_guard_pages();

/* 初始化 Java 语言的同步系统 */
ObjectMonitor::Initialize() ;

// Second phase of bootstrapping, VM is about entering multi-thread mode
MemTracker::bootstrap_multi_thread();

/* 初始化全局模块 */
jint status = init_globals();
if (status != JNI_OK) {
  delete main_thread;
  *canTryAgain = false; // don't let caller call JNI_CreateJavaVM again
  return status;
}
```

```
// Should be done after the heap is fully created
main_thread->cache_global_variables();

HandleMark hm;

{ MutexLocker mu(Threads_lock);
  Threads::add(main_thread);
}

// Any JVMTI raw monitors entered in onload will transition into
// real raw monitor. VM is setup enough here for raw monitor enter.
JvmtiExport::transition_pending_onload_raw_monitors();

// Fully start NMT
MemTracker::start();

// Create the VMThread
{ TraceTime timer("Start VMThread", TraceStartupTime);
  VMThread::create();
  Thread* vmthread = VMThread::vm_thread();

  if (!os::create_thread(vmthread, os::vm_thread))
    vm_exit_during_initialization("Cannot create VM thread.
                                   Out of system resources.");

  // Wait for the VM thread to become ready, and VMThread::run to initialize
  // Monitors can have spurious returns, must always check another state flag
  {
    MutexLocker ml(Notify_lock);
    os::start_thread(vmthread);
    while (vmthread->active_handles() == NULL) {
      Notify_lock->wait();
    }
  }
}

assert (Universe::is_fully_initialized(), "not initialized");
if (VerifyBeforeGC && VerifyGCStartAt == 0) {
  Universe::heap()->prepare_for_verify();
  Universe::verify();   // make sure we're starting with a clean slate
}


EXCEPTION_MARK;

// At this point, the Universe is initialized, but we have not executed
// any byte code.  Now is a good time (the only time) to dump out the
// internal state of the JVM for sharing.
```

```
  if (DumpSharedSpaces) {
    Universe::heap()->preload_and_dump(CHECK_0);
    ShouldNotReachHere();
  }

  // Always call even when there are not JVMTI environments yet,
       since environments
  // may be attached late and JVMTI must track phases of VM execution
  JvmtiExport::enter_start_phase();

  // Notify JVMTI agents that VM has started (JNI is up) - nop if no agents.
  JvmtiExport::post_vm_start();

  {
    TraceTime timer("Initialize java.lang classes", TraceStartupTime);

    if (EagerXrunInit && Arguments::init_libraries_at_startup()) {
      create_vm_init_libraries();
    }

    if (InitializeJavaLangString) {
      initialize_class(vmSymbols::java_lang_String(), CHECK_0);
    } else {
      warning("java.lang.String not initialized");
    }

    if (AggressiveOpts) {
      {
        // Forcibly initialize java/util/HashMap and mutate the private
        // static final "frontCacheEnabled" field before we start
             creating instances
#ifdef ASSERT
        klassOop tmp_k = SystemDictionary::find
          (vmSymbols::java_util_HashMap(), Handle(), Handle(), CHECK_0);
        assert(tmp_k == NULL, "java/util/HashMap should not be loaded yet");
#endif
        klassOop k_o = SystemDictionary::resolve_or_null
          (vmSymbols::java_util_HashMap(), Handle(), Handle(), CHECK_0);
        KlassHandle k = KlassHandle(THREAD, k_o);
        guarantee(k.not_null(), "Must find java/util/HashMap");
        instanceKlassHandle ik = instanceKlassHandle(THREAD, k());
        ik->initialize(CHECK_0);
        fieldDescriptor fd;
        // Possible we might not find this field; if so, don't break
        if (ik->find_local_field(vmSymbols::frontCacheEnabled_name(),
             vmSymbols::bool_signature(), &fd)) {
          k()->java_mirror()->bool_field_put(fd.offset(), true);
        }
      }
```

```
  if (UseStringCache) {
    // Forcibly initialize java/lang/StringValue and mutate the private
    // static final "stringCacheEnabled" field before we start
         creating instances
    klassOop k_o = SystemDictionary::resolve_or_null
     (vmSymbols::java_lang_StringValue(), Handle(), Handle(), CHECK_0);
    // Possible that StringValue isn't present: if so, silently don't break
    if (k_o != NULL) {
      KlassHandle k = KlassHandle(THREAD, k_o);
      instanceKlassHandle ik = instanceKlassHandle(THREAD, k());
      ik->initialize(CHECK_0);
      fieldDescriptor fd;
      // Possible we might not find this field: if so,
           silently don't break
      if (ik->find_local_field(vmSymbols::stringCacheEnabled_name(),
           vmSymbols::bool_signature(), &fd)) {
        k()->java_mirror()->bool_field_put(fd.offset(), true);
      }
    }
  }
}

/* 初始化 Java 的一些基础类库 */
if (InitializeJavaLangSystem) {
  initialize_class(vmSymbols::java_lang_System(), CHECK_0);
  initialize_class(vmSymbols::java_lang_ThreadGroup(), CHECK_0);
  Handle thread_group = create_initial_thread_group(CHECK_0);
  Universe::set_main_thread_group(thread_group());
  initialize_class(vmSymbols::java_lang_Thread(), CHECK_0);
  oop thread_object = create_initial_thread
       (thread_group, main_thread, CHECK_0);
  main_thread->set_threadObj(thread_object);
  // Set thread status to running since main thread has
  // been started and running.
  java_lang_Thread::set_thread_status(thread_object,
                                  java_lang_Thread::RUNNABLE);

  // The VM preresolve methods to these classes.
        Make sure that get initialized
  initialize_class(vmSymbols::java_lang_reflect_Method(), CHECK_0);
  initialize_class(vmSymbols::java_lang_ref_Finalizer(), CHECK_0);
  // The VM creates & returns objects of this class.
        Make sure it's initialized.
  initialize_class(vmSymbols::java_lang_Class(), CHECK_0);
  call_initializeSystemClass(CHECK_0);

  // get the Java runtime name after java.lang.System is initialized
  JDK_Version::set_runtime_name(get_java_runtime_name(THREAD));
```

```
        JDK_Version::set_runtime_version(get_java_runtime_version(THREAD));
      } else {
        warning("java.lang.System not initialized");
      }

        if (InitializeJavaLangExceptionsErrors) {
        initialize_class(vmSymbols::java_lang_OutOfMemoryError(),
CHECK_0);
        initialize_class(vmSymbols::java_lang_NullPointerException(),
CHECK_0);
        initialize_class(vmSymbols::java_lang_ClassCastException(),
CHECK_0);
        initialize_class(vmSymbols::java_lang_ArrayStoreException(),
CHECK_0);
        initialize_class(vmSymbols::java_lang_ArithmeticException(),
CHECK_0);
        initialize_class(vmSymbols::java_lang_StackOverflowError(),
CHECK_0);
        initialize_class(vmSymbols::java_lang_IllegalMonitorState-
Exception(), CHECK_0);
        initialize_class(vmSymbols::java_lang_IllegalArgumentException(),
CHECK_0);
      } else {
        warning("java.lang.OutOfMemoryError has not been initialized");
        warning("java.lang.NullPointerException has not been initialized");
        warning("java.lang.ClassCastException has not been initialized");
        warning("java.lang.ArrayStoreException has not been initialized");
        warning("java.lang.ArithmeticException has not been initialized");
        warning("java.lang.StackOverflowError has not been initialized");
        warning("java.lang.IllegalArgumentException has not been initialized");
      }
    }

    initialize_class(vmSymbols::java_lang_Compiler(), CHECK_0);

    // More hackery - the static initializer of java.lang.Compiler adds the
    // string "nojit" to the java.vm.info property if no jit gets loaded
    // through java.lang.Compiler (the hotspot compiler does not get loaded
    // through java.lang.Compiler).  "java -version" with the hotspot vm
    // says "nojit" all the time which is confusing.  So, we reset it here.
    // This should also be taken out as soon as 4211383 gets fixed.
    reset_vm_info_property(CHECK_0);

    quicken_jni_functions();

    // Must be run after init_ft which initializes ft_enabled
    if (TRACE_INITIALIZE() != JNI_OK) {
      vm_exit_during_initialization("Failed to initialize tracing backend");
    }
```

```
  // Set flag that basic initialization has completed. Used by exceptions
  // and various debug stuff, that does not work until all basic classes
  // have been initialized.
  set_init_completed();

#ifndef USDT2
  HS_DTRACE_PROBE(hotspot, vm__init__end);
#else /* USDT2 */
  HOTSPOT_VM_INIT_END();
#endif /* USDT2 */

  /* 记录虚拟机初始化完成时间 */
  Management::record_vm_init_completed();

  /* 指定类加载器 */
  SystemDictionary::compute_java_system_loader(THREAD);
  if (HAS_PENDING_EXCEPTION) {
    vm_exit_during_initialization(Handle(THREAD, PENDING_EXCEPTION));
  }

#ifndef SERIALGC
  // Support for ConcurrentMarkSweep. This should be cleaned up
  // and better encapsulated. The ugly nested if test would go away
  // once things are properly refactored. XXX YSR
  if (UseConcMarkSweepGC || UseG1GC) {
    if (UseConcMarkSweepGC) {
      ConcurrentMarkSweepThread::makeSurrogateLockerThread(THREAD);
    } else {
      ConcurrentMarkThread::makeSurrogateLockerThread(THREAD);
    }
    if (HAS_PENDING_EXCEPTION) {
      vm_exit_during_initialization(Handle(THREAD, PENDING_EXCEPTION));
    }
  }
#endif // SERIALGC

  // Always call even when there are not JVMTI environments yet, since
  // environments may be attached late and JVMTI must track phases of VM

  // execution
  JvmtiExport::enter_live_phase();

  /* 初始化 os 模块的信号系统，这样虚拟机才可以向 os 发送相应的信号信息 */
  os::signal_init();

  /* 启动 Attach Listener 线程 */
  if (!DisableAttachMechanism) {
    AttachListener::vm_start();
```

```
    if (StartAttachListener || AttachListener::init_at_startup()) {
      AttachListener::init();
    }
  }

  // Launch -Xrun agents
  // Must be done in the JVMTI live phase so that for backward compatibility
  // the JDWP back-end can launch with -Xdebug -Xrunjdwp.
  if (!EagerXrunInit && Arguments::init_libraries_at_startup()) {
    create_vm_init_libraries();
  }

  // Notify JVMTI agents that VM initialization is complete - nop if no agents.
  JvmtiExport::post_vm_initialized();

  if (TRACE_START() != JNI_OK) {
    vm_exit_during_initialization("Failed to start tracing backend.");
  }

  if (CleanChunkPoolAsync) {
    Chunk::start_chunk_pool_cleaner_task();
  }

  /* 初始化即时编译器 */
  CompileBroker::compilation_init();
  /* 初始化 Management 模块 */
  Management::initialize(THREAD);
  if (HAS_PENDING_EXCEPTION) {
    // management agent fails to start possibly due to
    // configuration problem and is responsible for printing
    // stack trace if appropriate. Simply exit VM.
    vm_exit(1);
  }

  if (Arguments::has_profile())FlatProfiler::engage(main_thread, true);
  if (Arguments::has_alloc_profile()) AllocationProfiler::engage();
  if (MemProfiling)                   MemProfiler::engage();
  StatSampler::engage();
  if (CheckJNICalls)                  JniPeriodicChecker::engage();

  BiasedLocking::init();

  if (JDK_Version::current().post_vm_init_hook_enabled()) {
    call_postVMInitHook(THREAD);
    // The Java side of PostVMInitHook.run must deal with all
    // exceptions and provide means of diagnosis.
    if (HAS_PENDING_EXCEPTION) {
      CLEAR_PENDING_EXCEPTION;
    }
```

```
    }

    {
        MutexLockerEx ml(PeriodicTask_lock, Mutex::_no_safepoint_check_flag);
        /* 启动 WatcherThread 线程，用以支持定时器等周期性任务 */
        WatcherThread::make_startable();
        // Start up the WatcherThread if there are any periodic tasks
        // NOTE:  All PeriodicTasks should be registered by now. If they
        //   aren't, late joiners might appear to start slowly (we might
        //   take a while to process their first tick).
        if (PeriodicTask::num_tasks() > 0) {
            WatcherThread::start();
        }
    }

    /* 额外 os 模块初始化的机会 */
    os::init_3();

    create_vm_timer.end();
    return JNI_OK;
  }
```

5.4 跟踪HotSpot的初始化过程

本书在 5.3.2 节中示例了 create_vm()函数的完整代码，那么接下来笔者就开始为大家讲解 HotSpot 在初始化过程中所经历的一些详细步骤。如下所示：

（1）首先执行版本检查；

（2）检查并确认是否支持当前的 JNI 版本，并调用 ostream_init()函数初始化输出流模块；

（3）调用 Arguments 模块的 process_sun_java_launcher_properties()函数配置 Launcher 的一些相关参数；

（4）调用 init()函数初始化 os 模块，包括：随机数生成器（Random Number Generator）、当前进程 id（Current Process id）、高精度计时器（High-Resolution Timer）、内存页尺寸（Memory Page Sizes），以及保护页（Guard Pages）等；

（5）初始化一些系统属性，比如：java.version、java.vendor 和 os.name 等；

（6）参数解析和记录虚拟机的创建时间等；

（7）在解析 args 参数后调用 os 模块的 init_2()函数对内存、栈、线程等与 os 模块密切相关的部分进行初始化；

（8）配置垃圾收集器的输出流模块；

（9）如果用到 agent 库（hprof、jdi）的话，则初始化并启动；

（10）初始化全局线程队列；

（11）初始化 TLS 模块；

（12）调用 vm_init_globals()函数初始化全局数据结构，包括：事件日志（Event Log）、os 同步原语、性能统计数据内存（perfMemory），以及内存分配器（chunkPool）等；

（13）创建 Java 主线程；

（14）初始化启动类加载器（Bootstrap ClassLoader）、CodeCache、解释器、JIT 编译器、JNI、系统词典（System Dictionary）等；

（15）将之前创建的 Java 主线程合并到线程队列中；

（16）创建 VM Thread；

（17）初始化 Java API 中的一些基础类库，比如 java.lang.String、java.lang.System、java.lang.Thread、java.lang.ThreadGroup、java.lang.reflect.Method、java.lang.ref.Finalizer、java.lang.Class，以及 java.lang.OutOfMemoryError 等 lang 包下的基础类库；

（18）启动 Signal Dispatcher 线程；

（19）初始化 JIT 编译器，并启动 HotSpot 编译代理线程；

（20）启动 HotSpot 中的一些辅助线程。

5.4.1 调用init()和init_2()函数初始化os模块

尽管 os 模块一共需要经历 3 次初始化过程，但是从严格意义上来说，init()和 init_2()函数才是 os 模块初始化的核心。那么第一次调用 init()函数是在 TLS 之前，而第二次调用 init_2()函数则是在 args 参数解析之后，也就是说，init()和 init_2()函数分别会在不同的时机对 os 模块进行初始化操作。

大家首先将目光转移到 os 模块的第一次初始化上，也就是调用 init()函数。当第一次调用 init()函数对 os 模块执行初始化时，该函数内部会初始化一些固定的系统配置。在 HotSpot 中，os 模块包含在/openjdk/hotspot/src/os 目录中，并且在该目录下又会根据操作系统的不同，划分有对应不同操作系统环境的子模块。本书以 Linux 平台为例，init()和 init_2()函数都包含在/openjdk/hotspot/src/os/linux/vm/os_linux.cpp 中。

在 init()函数内部，首先会设置 page 的大小。如下所示：

代码 5-4 init()函数的代码片段 1

```
Linux::set_page_size(sysconf(_SC_PAGESIZE));
if (Linux::page_size() == -1) {
  fatal(err_msg("os_linux.cpp: os::init: sysconf failed (%s)",
                strerror(errno)));
}
init_page_sizes((size_t) Linux::page_size());
```

当设置完 page 大小后，接下来就会调用 initialize_system_info()函数设置处理器数量、初始化处理器，以及获取实际的物理内存大小，并将内存大小保存在全局变量_physical_memory 中。如下所示：

代码 5-5 init()函数的代码片段 2

```
void os::Linux::initialize_system_info() {
  set_processor_count(sysconf(_SC_NPROCESSORS_CONF));
  if (processor_count() == 1) {
    pid_t pid = os::Linux::gettid();
    char fname[32];
    jio_snprintf(fname, sizeof(fname), "/proc/%d", pid);
    FILE *fp = fopen(fname, "r");
    if (fp == NULL) {
      unsafe_chroot_detected = true;
    } else {
      fclose(fp);
    }
  }
  _physical_memory = (julong)sysconf(_SC_PHYS_PAGES) *
      (julong)sysconf(_SC_PAGESIZE);
  assert(processor_count() > 0, "linux error");
}
```

当这些基本步骤执行完成之后，便会调用 pthread_self()函数获取原生主线程的句柄，并将其保存在全局变量_main_thread 中。在 init()函数的末尾，还会调用 clock_init()函数初始化系统时钟。

当 init()函数执行完成后，便会开始解析 args 参数，直至调用 init_2()函数继续对 os 模块进行初始化。和第一次初始化不同，第二次初始化主要是针对内存、栈、线程等与 os 模块密切相关的部分进行初始化。

在 init_2()的函数内部，首先会调用 fast_thread_clock_init()函数对快速线程时钟进行初始化，然后调用 mmap()函数分配共享内存，以及设置大页内存。接下来就会调用 SR_initialize()函数初始化内核信号，安装信号处理函数 SR_hanlder，用做线程执行过程中的 Suspended/Resumed 处理。如下所示：

代码 5-6 SR_initialize()函数的完整代码

```
static int SR_initialize() {
  struct sigaction act;
  char *s;
  /* Get signal number to use for suspend/resume */
  if ((s = ::getenv("_JAVA_SR_SIGNUM")) != 0) {
    int sig = ::strtol(s, 0, 10);
```

```
      if (sig > 0 || sig < _NSIG) {
          SR_signum = sig;
      }
    }

    assert(SR_signum > SIGSEGV && SR_signum > SIGBUS,
          "SR_signum must be greater than max(SIGSEGV, SIGBUS), see 4355769");

    sigemptyset(&SR_sigset);
    sigaddset(&SR_sigset, SR_signum);

    /* Set up signal handler for suspend/resume */
    act.sa_flags = SA_RESTART|SA_SIGINFO;
    act.sa_handler = (void (*)(int)) SR_handler;

    // SR_signum is blocked by default.
    // 4528190 - We also need to block pthread restart signal (32 on all
    // supported Linux platforms). Note that LinuxThreads need to block
    // this signal for all threads to work properly. So we don't have
    // to use hard-coded signal number when setting up the mask.
    pthread_sigmask(SIG_BLOCK, NULL, &act.sa_mask);

    if (sigaction(SR_signum, &act, 0) == -1) {
      return -1;
    }

    // Save signal flag
    os::Linux::set_our_sigflags(SR_signum, act.sa_flags);
    return 0;
  }
```

当初始化内核信号结束后，紧接着就会对线程栈进行一系列的配置，比如设置线程栈大小、分配线程初始栈等。如下所示：

代码 5-7　init_2()函数的代码片段

```
  os::Linux::min_stack_allowed = MAX2(os::Linux::min_stack_allowed,
           (size_t)(StackYellowPages+StackRedPages+StackShadowPages) *
           Linux::page_size() + (2*BytesPerWord COMPILER2_PRESENT(+1)) *
           Linux::vm_default_page_size());

  size_t threadStackSizeInBytes = ThreadStackSize * K;
  if (threadStackSizeInBytes != 0 &&
      threadStackSizeInBytes < os::Linux::min_stack_allowed) {
        tty->print_cr("/nThe stack size specified is too small, "
                      "Specify at least %dk",
                      os::Linux::min_stack_allowed/ K);
        return JNI_ERR;
```

```
}

// Make the stack size a multiple of the page size so that
// the yellow/red zones can be guarded.
JavaThread::set_stack_size_at_create(round_to(threadStackSizeInBytes,
     vm_page_size()));

Linux::capture_initial_stack(JavaThread::stack_size_at_create());
```

当线程栈配置完成后，就会调用 set_createThread_lock()函数初始化时钟，以及调用 prio_init()函数初始化线程优先级策略等。经过这一系列的步骤之后，init_2()函数则完成了对 os 模块的第二次初始化。

5.4.2 调用vm_init_globals()函数初始化全局数据结构

包含在/openjdk/hotspot/src/share/vm/runtime/init.cpp 中的 vm_init_globals()函数实现了对全局数据结构的初始化，这些全局数据结构主要包括：Java 的原始数据类型、分配全局时间缓存区、初始化全局锁、初始化 ChunkPool 和 Perf Data 等。如下所示：

代码 5-8 vm_init_globals()函数的完整代码

```
void vm_init_globals() {
  check_ThreadShadow();
  basic_types_init();
  eventlog_init();
  mutex_init();
  chunkpool_init();
  perfMemory_init();
}
```

5.4.3 调用init_globals()函数初始化全局模块

init_globals()函数实现了对全局模块的初始化，这些全局模块都属于 HotSpot 的功能核心，主要包括 JMX 的 management 模块、Code Cache 模块、StubRoutines 模块、Universe 模块、解释器模块、模板表模块和 stubs 模块等。为了使大家阅读更方便，本书示例了包含在/openjdk/hotspot/src/share/vm/runtime/init.cpp 中 init_globals()函数的完整代码。如下所示：

代码 5-9 init_globals()函数的完整代码

```
jint init_globals() {
  HandleMark hm;
  /* 初始化 JMX 的 management 模块 */
  management_init();
  /* 初始化 bytecodes 模块 */
```

```
    bytecodes_init();
    /* 初始化类装载器模块 */
    classLoader_init();
    /* 初始化代码高速缓存模块 */
    codeCache_init();
    VM_Version_init();
    os_init_globals();
    /* 初始化 stubRoutines 模块的第 1 阶段 */
    stubRoutines_init1();
    /* 初始化 universe 模块 */
    jint status = universe_init();
    if (status != JNI_OK)
      return status;

    /* 初始化解释器模块 */
    interpreter_init();
    invocationCounter_init();  // before any methods loaded
    marksweep_init();
    accessFlags_init();
    /* 初始化模板表模块 */
    templateTable_init();
    InterfaceSupport_init();
    SharedRuntime::generate_stubs();
    universe2_init();  // dependent on codeCache_init and stubRoutines_init1
    referenceProcessor_init();
    jni_handles_init();
    vmStructs_init();

    vtableStubs_init();
    InlineCacheBuffer_init();
    compilerOracle_init();
    compilationPolicy_init();
    compileBroker_init();
    VMRegImpl::set_regName();

    if (!universe_post_init()) {
      return JNI_ERR;
    }
    javaClasses_init();   // must happen after vtable initialization
    /* 初始化 stubRoutines 模块的第 2 阶段 */
    stubRoutines_init2();

    // All the flags that get adjusted by VM_Version_init and os::init_2
    // have been set so dump the flags now.
    if (PrintFlagsFinal) {
      CommandLineFlags::printFlags(tty, false);
    }

    return JNI_OK;
  }
```

5.5 本章小结

本章笔者深入源码剖析了 HotSpot 的核心组成模块以及其初始化过程，其中涵盖了 os 模块的初始化、全局数据结构和全局模块的初始化等。那么在下一个章节中，笔者将会带领大家一起走进 HotSpot 的运行时内存区，为大家详细介绍 JVM 的运行时内存区以及自动内存管理机制，让大家彻底弄清楚对象所占用的内存空间究竟应该如何分配，在哪里分配，以及分配后如何执行回收。

第 6 章

内存分配与垃圾回收

在刚开始学习 Java 的时候，老师都应该告诉过我们，使用 Java 编写程序将再也不必要像 C/C++那样手动管理对象的内存分配与回收。不过在看似美好的背后，其实暗藏着“陷阱”，如果大家对 JVM 的自动内存管理机制不了解的话，那么这将会是一场灾难。本章笔者将会为大家详细介绍 JVM 的运行时内存区以及自动内存管理机制，让大家彻底弄清楚对象所占用的内存空间究竟应该如何分配，在哪里分配，以及分配后如何执行回收。

6.1 JVM的运行时内存区结构

内存在计算机世界里占据着至关重要的地位，任何运行时的程序或者数据都需要依靠内存作为存储介质，否则程序将无法正常运行。与 C/C++语言相比，使用 Java 语言编写的程序并不需要显式地为每一个对象都编写对应的内存分配和内存回收等相关函数，这主要是得益于 JVM 的自动内存管理机制，使得 Java 开发人员可以从繁琐的体力劳动中解放出来，只需关注于自身业务即可。

尽管JVM的自动内存管理机制大大提升了Java开发人员的编程效率，甚至从某种意义上来说还降低了内存泄露和内存溢出的风险，但是如果Java开发人员过度依赖于“自动”，那么这将会是一场灾难，最严重的就是会弱化Java开发人员在程序出现内存溢出时定位问题和解决问题的能力。在此需要提醒大家，千万不要被JVM的自动内存管理机制被动地牵着鼻子走，所以了解JVM究竟是如何实现自动分配内存和回收内存的，就显得非常重要，只有当大家真正了解JVM如何管理内存后，才能够在遇见OutOfMemoryError时，快速地根据错误异常日志定位问题和解决问题。参考《Java虚拟机规范 Java SE7 版》的描述来看，JVM内部

定义了多个程序在运行时需要使用到的内存区，如图 6-1 所示[①]。

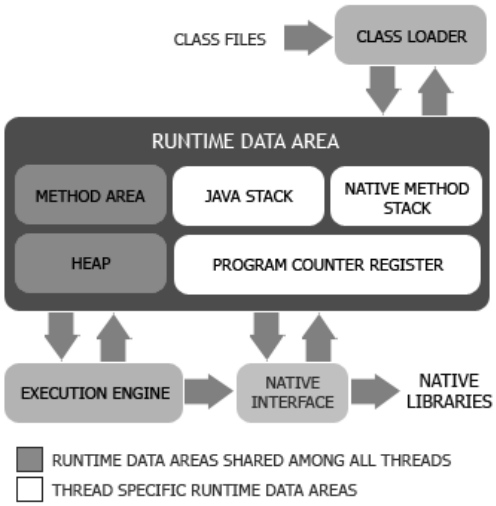


图 6-1 JVM 运行时内存区

JVM 的设计者们之所以会选择将 JVM 的内存结构划分为多个不同的内存区，是因为每一个独立的内存区都拥有各自的用途，都会负责存储各自的数据类型。其中一些内存区的生命周期往往还会和 JVM 的生命周期保持一致，也就是说，会伴随着 JVM 的启动而创建，伴随着 JVM 的退出而销毁。而另一部分内存区则是与线程的生命周期保持一致，会伴随着线程的开始而创建，伴随着线程的消亡而销毁。尽管不同的内存区在存储类型和生命周期上有一定区别，却都拥有一个相同的本质，那就是存储程序的运行时数据。

6.2 线程共享内存区

如图 6-1 所示，JVM 中的内存区可以根据受访权限的不同定义为线程共享和线程私有两大类。所谓线程共享指的就是可以允许被所有的线程共享访问的一类内存区，包括堆区、方法区和运行时常量池三个内存区。

6.2.1 Java堆区

相信对于绝大多数的 Java 开发人员而言，Java 堆区应该算得上是最为熟悉和平时接触最多的内存区之一，甚至有不少的 Java 开发人员还会非常粗粒度地将 JVM 中的内存区理解为仅有 Java 堆（heap）和 Java 栈（stack），尽管这种理解和划分非常不全面，但是从某种

① 图片来源于：http://www.blogjava.net/images/blogjava_net/nkjava/jvmstructure.png。

意义上来说却恰恰反映出了这 2 个内存区是绝大多数 Java 开发人员最关注的。Java 堆区在 JVM 启动的时候被创建，并且它在实际的内存空间中可以是不连续的。

Java 堆区是一块用于存储对象实例的内存区，同时也是 GC（Garbage Collection，垃圾收集器）执行垃圾回收的重点区域，正是因为 Java 堆区是 GC 的重点回收区域，那么 GC 极有可能会在大内存的使用和回收上成为性能瓶颈。为了解决这个问题，JVM 的设计者们开始考虑是否一定需要将对象实例存储到 Java 堆区内。笔者在 1.5.2 节中曾经介绍过基于 OpenJDK 深度定制的 TaoBaoVM，其中创新的 GCIH（GC invisible heap）技术实现 off-heap，将生命周期较长的 Java 对象从 heap 中移至 heap 之外，并且 GC 不能管理 GCIH 内部的 Java 对象，以此达到降低 GC 的回收平率和提升 GC 的回收效率的目的。除此之外，逃逸分析与栈上分配等优化技术（请阅读 6.5.2 节）同样也是降低 GC 回收频率和提升 GC 回收效率的一种方式，这样一来，Java 堆区就将不再是 Java 对象内存分配的唯一选择了。

存储在 JVM 中的 Java 对象可以被划分为两类：一类是生命周期较短的瞬时对象，这类对象的创建和消亡都非常迅速，而另外一类对象的生命周期却非常长，在某些极端的情况下还能够与 JVM 的生命周期保持一致。因此对于这些不同生命周期的 Java 对象，应该采取不同的垃圾收集策略，分代收集由此诞生。目前几乎所有的 GC 都是用分代收集算法，所以 Java 堆区如果要更进一步细分的话，还可以划分为新生代（YoungGen）和老年代（OldGen），其中新生代又可以划分为 Eden 空间、From Survivor 空间和 To Survivor 空间，如图 6-2 所示。

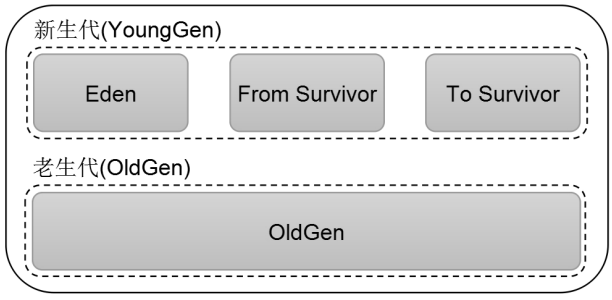


图 6-2　Java 堆区中的分代

既然 Java 堆区用于存储 Java 对象实例，那么堆的大小在 JVM 启动时就已经设定好了，大家可以通过选项“-Xmx”和“-Xms”来进行设置。其中选项“-Xmx”用于表示堆区的起始内存，而选项“-Xmx”则用于表示堆区的最大内存。一旦堆区中的内存大小超过“-Xmx”所指定的最大内存时，将会抛出 OutOfMemoryError 异常。

6.2.2 方法区

方法区和 Java 堆区一样，同样也是允许被所有的线程共享访问。方法区中存储了每一个 Java 类的结构信息，比如：运行时常量池、字段和方法数据、构造函数和普通方法的字节码内容以及类、实例、接口初始化时需要用到的特殊方法等数据。尽管 Java 虚拟机规范对方法区的具体实现方式并没有明确要求，但是在 HotSpot 中，方法区仅仅只是逻辑上的独立，实际上还是包含在 Java 堆区内，也就是说，方法区在物理上当然属于 Java 堆区的一部分。

方法区在 JVM 启动的时候被创建，并且它在实际的内存空间中和 Java 堆区一样都可以是不连续的。方法区是一块比较特殊的运行时内存区，有一些 Java 开发人员似乎更愿意将方法区称之为永久代（Permanent Generation），这主要是因为方法区除了可以通过选项“-XX:MaxPermSize”设置内存大小进行动态扩展外，并不会像 Java 堆区那样频繁地被 GC 执行回收，甚至还可以显式地指定是否需要在程序运行时回收方法区中的数据，这就是方法区被大家称之为永久代的真正原因。之前也说过了，尽管方法区并不会像 Java 堆区那么频繁地被执行内存回收，但是这并不代表方法区中的数据就永远不会被回收，如果在没有显式要求不对方法区进行内存回收的情况下，GC 的回收目标仅针对方法区中的常量池和类型卸载。

方法区同样也有可能发生内存溢出，一旦方法区中的内存大小超过选项“-XX:MaxPermSize”所指定的最大内存时，将会抛出 OutOfMemoryError 异常。

6.2.3 运行时常量池

运行时常量池属于方法区中的一部分，一个有效的字节码文件中除了包含类的版本信息、字段、方法以及接口等描述信息外，还包含一项信息那就是常量池表（Constant Pool Table），那么运行时常量池就是字节码文件中常量池表的运行时表示形式。运行时常量池中包含多种不同的常量，比如编译期就已经明确的数值字面量到运行期解析后才能够获得的方法或者字段引用。运行时常量池类似于传统编程语言中的符号表（symbol table），但是它所包含的数据却比符号表要更加丰富一些。

当类装载器成功将一个类或者接口装载进 JVM 后，就会创建与之对应的运行时常量池。在此大家需要注意，由于每一个运行时常量池所分配的内存来源于方法区，一旦所需要的内存大小超过方法区所能够提供的最大值时，运行时常量池同样也会抛出 OutOfMemoryError 异常。

6.3 线程私有内存区

和线程共享内存区不同，线程私有内存区是不允许被所有线程共享访问的。线程私有内存区是只允许被所属的独立线程进行访问的一类内存区，包括 PC 寄存器、Java 栈及本地方

法栈 3 个内存区。

6.3.1 PC寄存器

由于 JVM 是基于栈的架构，所以任何的操作都需要经过入栈和出栈来完成，在此大家需要注意，本书所提及的 JVM 中的 PC 寄存器（Program Counter Register）并非是广义上所指的物理寄存器，或许将其翻译为 PC 计数器会更加贴切，并且也不容易引起一些不必要的误会。JVM 中的 PC 寄存器是对物理 PC 寄存器的一种抽象模拟，它是线程私有的，生命周期与线程的生命周期保持一致。如果当前线程所执行的方法是一个 Java 方法，那么 PC 寄存器就会存储正在执行的字节码指令地址，反之如果是 native 方法，这时 PC 寄存器的值就是空（undefined）。

大家首先思考一下，PC 寄存器为什么会被设定为线程私有？我们都知道所谓的多线程在一个特定的时间段内只会执行其中某一个线程的方法，CPU 会不停地做任务切换，那么为了能够准确地记录各个线程正在执行的当前字节码指令地址，最好的办法自然是为每一个线程都分配一个 PC 寄存器，这样一来各个线程之间便可以进行独立计算，从而不会出现相互干扰的情况。尽管明确了 PC 寄存器的作用和目的，但是存储字节码指令地址又有什么用呢？JVM 的字节码解释器就需要通过改变 PC 寄存器的值来明确下一条应该执行什么样的字节码指令，当然 Java 虚拟机规范并没有明确要求一定要采用这种方式去实现。PC 寄存器是 JVM 的内存区中唯一一个没有明确规定需要抛出 OutOfMemoryError 异常的运行时内存区。

6.3.2 Java栈

在 Java 虚拟机规范中，Java 栈也可以被称之为 Java 虚拟机栈（Java Virtual Machine stack），它同 PC 寄存器一样都是线程私有，并且生命周期与线程的生命周期保持一致。Java 栈用于存储栈帧（Stack Frame），而栈帧中所存储的就是局部变量表、操作数栈，以及方法出口等信息，关于栈帧本书则会在第 8 章中进行详细讲解。

在 6.2.1 节中，笔者曾经提及过 Java 堆区和 Java 栈是许多开发人员最关心的两个运行时内存区。Java 堆区中既然存储的是对象实例，那么 Java 栈中的局部变量表就是用于存储各类原始数据类型、对象引用（reference）以及 returnAddress 类型。参考《Java 虚拟机规范 Java SE7 版》的描述来看，returnAddress 类型被定义为 Java 虚拟机内部的原始数据类型，该类型用于表示一条字节码指令的操作码（opcode）。但 returnAddress 类型在 Java 语言中却并不存在相对应的类型，同时自然也无法在运行时更改 returnAddress 类型的值。尽管开发人员无法在程序中直接使用 returnAddress 类型，但在 Java7 之前，该类型却被用于 finally 子句的实现。

Java 栈允许被实现成固定大小的内存或者是可动态扩展的内存大小，在此需要提醒大家，如果 Java 栈被设定为固定大小的内存，一旦线程请求分配的栈容量超过 JVM 所允许的最大值时，JVM 将会抛出一个 StackOverflowError 异常，反之抛出一个 OutOfMemoryError 异常。

6.3.3 本地方法栈

本地方法栈（Native Method Stack）用于支持本地方法（native 方法，比如使用 C/C++ 代码编写的方法）的执行，它和 Java 栈的作用类似。Java 虚拟机规范并没有明确要求本地方法栈的具体实现方式，甚至如果 JVM 产品并不打算支持 native 方法，也不依赖于传统栈，则可以无需实现本地方法栈。不过一旦 JVM 中实现有本地方法栈时，那么它将会和 Java 栈一样，允许被实现成固定或者是可动态扩展的内存大小，并且本地方法栈同样也会抛出 StackOverflowError 或者 OutOfMemoryError 异常。

6.4 性能监控区

笔者曾经在 4.2.4 小节中介绍过 Prims 模块的子模块 PERF，在 Prims 模块下提供多个允许被外部程序访问 JVM 内部信息的对外访问接口，其中 PERF 模块是 sun.misc.Perf 类的底层实现，函数名称都是以“PERF_”进行开头，用于监控 JVM 内部的 Perf Data 计数器。简单来说，性能监控区就是 JVM 提供的一块内存共享区，专供外部程序访问这块区域中的 Perf Data（性能监控数据），以此实现外部程序监控 JVM 的性能指标，不过 Java 虚拟机规范并没有明确要求一定要实现性能监控区。

6.5 自动内存管理

程序内存管理其实是一个非常敏感的话题，对于 C/C++开发人员而言，他们可以在语法层面随意控制程序中一个对象的生命周期，即能够自由宣布对象诞生，又能够随时宣判对象死亡，因此我们将这种方式称之为手动内存管理。尽管手动内存管理非常自由与灵活，但是其弊端同样也非常明显，由于手动内存管理所具备的复杂性，在某些情况下往往会直接或者间接导致程序在运行过程中崩溃，并且一旦出现这种情况，开发人员便会将大把的时间浪费在问题定位上，所以手动内存管理是一把利与弊同样都非常明显的双刃剑。

手动内存管理究竟会发生哪些意外终止程序的正常运行呢？其实综合来看无非也就是内存溢出和内存泄露这两个最主要的原因。内存泄露也称之为存储渗漏，程序中发生内存泄露的一个比较常见的场景就是当你打算释放一个链表所引用的所有空间时，却错误地只释放

了链表的第一个元素，而剩下的元素尽管已经不再被引用，但是它们却离开了整个程序的控制范围，这样一来，链表中的元素所占用的内存空间将永远无法被释放，这就是内存泄露。尽管内存泄露并不会立刻引起程序崩溃，但是一旦发生内存泄露，程序中的可用内存就会被逐步蚕食，直至耗尽所有内存，最终导致程序崩溃。而内存溢出相对于内存泄露来说，尽管更容易被理解，但是同样的，内存溢出也是引发程序崩溃的罪魁祸首之一。或许手动内存管理所具备的便捷性和灵活性在某些情况下显得非常方便，但是这却无法确保每一个开发人员都能够在创建一个新对象时记得在使用完成之后释放掉其所占用的内存空间。因此无论你的工作年限有多长，技术能力有多强，内存溢出的问题都有可能悄悄地嵌入在你编写的代码中，而你却仍然浑然不觉，直到程序中的可用内存变得越来越少，甚至少到已经无法再为新对象分配内存空间引发程序崩溃之时。

在简短介绍了基于手动内存管理可能会导致程序出现崩溃的潜在原因后，接下来我们再来了解 JVM 的自动内存管理机制。自动内存管理简单来说就是无需开发人员手动参与内存的分配与回收，这样不仅能够降低内存泄露和内存溢出的风险，更重要的是自动内存管理机制能够使得开发人员更关注于自身业务。但是对于 Java 开发人员而言，自动内存管理就像是一个黑匣子，如果过度依赖于“自动”，那么这将会是一场灾难，最严重的就会弱化 Java 开发人员在程序出现内存溢出时定位问题和解决问题的能力。所以了解 JVM 的自动内存分配和内存回收原理就显得非常重要，只有在真正了解 JVM 是如何管理内存后，我们才能够在遇见 OutOfMemoryError 时，快速地根据错误异常日志定位问题和解决问题。在高级编程语言中，自动内存管理机制无疑已经是未来的发展趋势，所以除了 Java 采用了自动内存管理机制外，还有像 Lisp、C#、Python、Ruby 等编程语言同样也都采用了自动内存管理机制来实现内存的动态分配和垃圾回收操作。

6.5.1 内存分配原理

尽管 Java 对象的内存分配可以选择在堆外进行，但是不可否认这仅仅只是为了降低 GC 回收频率以及提升 GC 回收效率的一种辅助手段，所以 Java 堆区仍然是分配/存储对象实例的主要区域，这一点必然是毋庸置疑的。参考《Java 虚拟机规范 Java SE7 版》的描述来看，JVM 中包含三种引用类型，分别是类类型（class type）、数组类型（array type）和接口类型（interface type），这些引用类型的值则分别由类实例、数组实例以及实现了某个接口的派生类实例负责动态创建，那么 JVM 中究竟是如何为这些类型创建对应的对象实例呢？本书以创建一个普通的 Java 对象为例，如果是在 Java 语法层面上创建一个对象无非就是使用一个简单的 new 关键字即可，但是在 JVM 中就没有这么简单了，其中牵扯到的细节相当复杂而且过程繁多。简单来说，当语法层面使用 new 关键字创建一个 Java 对象时，JVM 首先会检查这个 new 指令的参数能否在常量池中定位到一个类的符号引用，然后检查与这个符号引用相对应的类是否已经成功经历过加载、解析和初始化等步骤（请阅读第 7 章），当类

完成装载步骤之后，就已经完全可以确定出创建对象实例时所需要的内存空间大小，接下来JVM 将会对其进行内存分配，以存储所生成的对象实例。如图 6-3 所示。

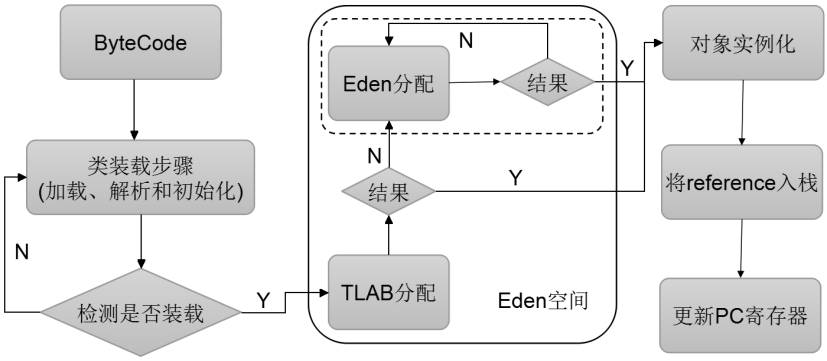


图 6-3 对象内存分配

为新对象分配内存是一件非常严谨和复杂的任务，JVM 的设计者们不仅需要考虑内存如何分配、在哪里分配等问题，并且由于内存分配算法与内存回收算法（请阅读 6.5.6 节）密切相关，所以还需要考虑 GC 执行完内存回收后是否会在内存空间中产生内存碎片。如果内存空间以规整和有序的方式分布，即已用和未用的内存都各自一边，彼此之间维系着一个记录下一次分配起始点的标记指针，当为新对象分配内存时，只需要通过修改指针的偏移量将新对象分配在第一个空闲内存位置上，这种分配方式就叫做指针碰撞（Bump the Pointer），反之则只能使用空闲列表（Free List）执行内存分配。

基于分代的概念，Java 堆区如果还要更进一步细分的话，还可以划分为新生代（YoungGen）和老年代（OldGen），其中新生代内又可以划分为 Eden 空间、From Survivor 空间和 To Survivor 空间。那么对象实例究竟是存储在堆区中的哪一个区域下呢？相信大家已经知道在 JVM 的运行时数据区中，堆区和方法区是线程共享区域，任何线程都可以访问到这两个区域中的共享数据，由于对象实例的创建在 JVM 中非常频繁，因此在并发环境下从堆区中划分内存空间是非线程安全的，所以务必需要保证数据操作的原子性。基于线程安全的考虑，如果一个类在分配内存之前已经成功完成类装载步骤之后，JVM 就会优先选择在 TLAB（Thread Local Allocation，本地线程分配缓冲区）中为对象实例分配内存空间，TLAB 在 Java 堆区中是一块线程私有区域，它包含在 Eden 空间内，除了可以避免一系列的非线程安全问题外，同时还能够提升内存分配的吞吐量，因此我们可以将这种内存分配方式称之为快速分配策略。

尽管不是所有的对象实例都能够在 TLAB 中成功分配内存，但 JVM 确实是将 TLAB 作为内存分配的首选，在程序中，开发人员可以通过选项“-XX:UseTLAB”设置是否开启 TLAB 空间。在此大家需要注意，TLAB 空间的内存非常小，缺省情况下仅占有整个 Eden 空间的

1%，当然我们可以通过选项“-XX:TLABWasteTargetPercent”设置 TLAB 空间所占用 Eden 空间的百分比大小。一旦对象在 TLAB 空间分配内存失败时，JVM 就会尝试着通过使用加锁机制确保数据操作的原子性，从而直接在 Eden 空间中分配内存，如果当在 Eden 空间中也无法分配内存时，JVM 就会执行 Minor GC，直至最终可以在 Eden 空间中分配内存为止（如果是大对象则直接在老年代中分配）。为了能够更好地理解对象内存空间的分配原理，本书示例了包含在/openjdk/hotspot/src/share/vm/interpreter/bytecodeInterpreter.cpp 中字节码解释器的代码片段（更多的情况下，官方使用的却是 interpreter 模块下的模板解释器）。如下所示：

代码 6-1　bytecodeInterpreter.cpp 的代码片段 1

```
  if (!constants->tag_at(index).is_unresolved_klass()) {
    // Make sure klass is initialized and doesn't have a finalizer
    oop entry = constants->slot_at(index).get_oop();
    assert(entry->is_klass(), "Should be resolved klass");
    klassOop k_entry = (klassOop) entry;
    assert(k_entry->klass_part()->oop_is_instance(),        "Should        be
instanceKlass");
    instanceKlass* ik = (instanceKlass*) k_entry->klass_part();
    if ( ik->is_initialized() && ik->can_be_fastpath_allocated() ) {
      size_t obj_size = ik->size_helper();
      oop result = NULL;
      // If the TLAB isn't pre-zeroed then we'll have to do it
      bool need_zero = !ZeroTLAB;
      /* 检测是否需要在 TLAB 中分配内存 */
      if (UseTLAB) {
        result = (oop) THREAD->tlab().allocate(obj_size);
      }
      if (result == NULL) {
        need_zero = true;
        // Try allocate in shared eden
  retry:
        /* 如果在 TLAB 中分配内存失败时，则直接在 Eden 中分配内存 */
        HeapWord* compare_to = *Universe::heap()->top_addr();
        HeapWord* new_top = compare_to + obj_size;
        if (new_top <= *Universe::heap()->end_addr()) {
          if (Atomic::cmpxchg_ptr(new_top, Universe::heap()->top_addr(),
              compare_to) != compare_to) {
            goto retry;
          }
          result = (oop) compare_to;
        }
      }
  if (result != NULL) {
    /* 对分配后的内存空间进行零值初始化 */
    if (need_zero ) {
      HeapWord* to_zero = (HeapWord*) result + sizeof(oopDesc) / oopSize;
```

```
      obj_size -= sizeof(oopDesc) / oopSize;
      if (obj_size > 0 ) {
        memset(to_zero, 0, obj_size * HeapWordSize);
      }
    }
      if (UseBiasedLocking) {
        result->set_mark(ik->prototype_header());
      } else {
        result->set_mark(markOopDesc::prototype());
      }
      result->set_klass_gap(0);
      result->set_klass(k_entry);
      /* 将对象引用入栈 */
      SET_STACK_OBJECT(result, 0);
      /* 更新 PC 寄存器中的字节码指令地址 */
      UPDATE_PC_AND_TOS_AND_CONTINUE(3, 1);
    }
  }
}
```

当为对象成功分配好所需的内存空间后，JVM 接下来要做的事情就是初始化对象实例。JVM 首先会对分配后的内存空间进行零值初始化，这一步操作确保了对象的实例字段在 Java 代码中可以不用赋初始值就能够直接使用，程序能够访问到这些字段的数据类型所对应的零值。本书示例了包含在/openjdk/hotspot/src/share/vm/interpreter/bytecodeInterpreter.cpp 中字节码解释器的代码片段。如下所示：

代码 6-2 bytecodeInterpreter.cpp 的代码片段 2

```
/* 将分配后的内存空间进行零值初始化 */
if (need_zero ) {
  HeapWord* to_zero = (HeapWord*) result + sizeof(oopDesc) / oopSize;
  obj_size -= sizeof(oopDesc) / oopSize;
  if (obj_size > 0 ) {
    memset(to_zero, 0, obj_size * HeapWordSize);
  }
}
```

对分配后的内存空间进行零值初始化后，JVM 就会初始化对象头和实例数据（在 HotSpot 中，内存空间中所存储的对象信息主要包含这两个部分），最后将对象引用入栈后再更新 PC 寄存器中的字节码指令地址。经过这一系列的操作步骤之后，一个 Java 对象实例才算是真正创建成功。

6.5.2 逃逸分析与栈上分配

之前曾经提及过，Java 堆区已经不再是对象内存分配的唯一选择，如果希望降低 GC 的回收平率和提升 GC 的回收效率，那么则可以使用堆外存储技术。目前最常见的堆外存储技术就是利用逃逸分析技术筛选出未发生逃逸的对象，然后避开堆区而直接选择在栈帧中分配内存空间。

逃逸分析（Escape Analysis）是 JVM 在执行性能优化之前的一种分析技术，它的具体目标就是分析出对象的作用域。简单来说，当一个对象被定义在方法体内部之后，它的受访权限仅限于方法体内，一旦其引用被外部成员引用后，这个对象就因此发生了逃逸，反之如果定义在方法体内的对象并没有被任何的外部成员引用时，JVM 就会为其在栈帧中分配内存空间，如下所示：

代码 6-3 栈上分配示例

```
/**
 * 逃逸分析与栈上分配
 *
 * @author JohnGao
 */
public class StackAllocation {
  public StackAllocation obj;
  public StackAllocation getStackAllocation() {
      /* 方法返回 StackAllocation 对象实例，发生逃逸 */
      return null == obj ? new StackAllocation() : obj;
  }
  public void setStackAllocation() {
      /* 为成员变量进行赋值，发生逃逸 */
      obj = new StackAllocation();
  }
  public void useStackAllocation1() {
      /* 引用成员变量的值，发生逃逸. */
      StackAllocation obj = getStackAllocation();
  }
  public void useStackAllocation2() {
      /* 对象的作用域仅限于方法体内，未发生逃逸. */
      StackAllocation obj = new StackAllocation();
  }
}
```

由于对象直接在栈上分配内存，因此 GC 就无需执行垃圾回收。栈帧会伴随着方法的调用而创建，伴随着方法的执行结束而销毁，由此可见，栈上分配的对象所占用的内存空间将会随着栈帧的出栈而释放。在 JDK 6u23 版本之后，HotSpot 中默认就已经开启了逃逸分析，

如果使用的是较早的版本，开发人员则可以通过选项“-XX:+DoEscapeAnalysis”显式开启逃逸分析，以及通过选项“-XX:+PrintEscapeAnalysis”查看逃逸分析的筛选结果。

6.5.3 对象内存布局与OOP-Klass模型

当成功对分配后的内存空间执行零值初始化后，JVM 接下来就会对对象进行实例化。在 HotSpot 中，对象实例化操作无非就是初始化对象头和实例数据，而且存储对象实例信息的内存布局也主要由这两个部分构成，如图 6-4 所示。先从对象头开始谈起，对象头中主要用于存储 Mark Word 和元数据指针等数据，其中 Mark Word 主要用于存储对象运行时的数据信息，比如 HashCode、GC 分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等。而元数据指针则是用于指向方法区中目标类的类型信息，也就是说通过元数据指针可以准确定位到当前对象的具体目标类型。

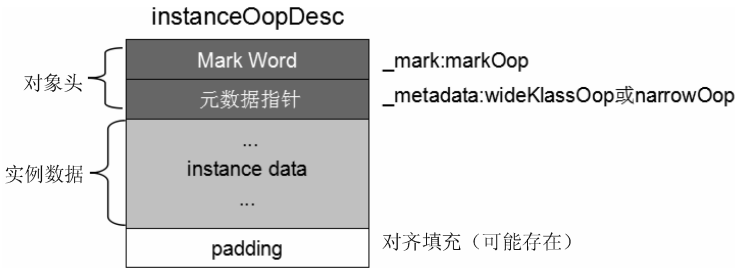


图 6-4 对象内存布局

除了对象头之外，内存布局的另一部分就是实例数据。实例数据主要用于存储定义在当前对象中各种类型的字段信息（包括派生于超类的字段信息），存储在实例对象中的字段顺序除了会与字段在 Java 类中定义的顺序有关外，还会受到 JVM 分配策略参数（FieldsAllocationStyle）的影响，当然开发人员可以通过选项“-XX:FieldsAllocationStyle”设置 JVM 的分配策略参数。HotSpot 缺省会按照 longs/doubles、ints、shorts/chars、bytes/booleans、oops 的分配策略顺序进行分配，相信大家仔细观察 HotSpot 缺省的分配策略后会发现，二进制位数相同的字段总是被划分到一起。在此大家需要注意，在满足这个前提的情况下，在超类中定义的变量很有可能会出现在派生类之前。

当大家理解 Java 对象在内存中是如何存储的后，接下来我们再来看看 JVM 中究竟是如何表达 Java 类以及对象实例的，毕竟 Java 语言只是一个中间语言，运行在 JVM 中还是需要一套完整的底层内部对象表示机制。OOP-Klass 模型就是用于表示 Java 类以及对象实例的一种数据结构，其中 OOP（Ordinary Object Pointer，普通对象指针）用于描述对象的实例信息，而 Klass 则用于描述对象实例的目标类型，也就是说 Klass 其实是一个与 Java 类相对应的 JVM 中的内部对等体，我们也可以将其称之为 C++对等体。

OOP 与 Klass 其实是两个相互独立但是却又彼此相互关联的模块，这两个模块均包含在/openjdk/hotspot/src/share/vm/oops 模块中，那么 OOP-Klass 模型与对象的内存布局之间又有什么关系呢？在 JVM 中对象头就是由 OOP 对象 instanceOopDesc 来表示的（数组类型则用 arrayOopDesc 对象来表示），而对象头中的元数据指针所指向的当前对象的目标类型则是由 Klass 中的 instanceKlass 对象表示的（数组则用 arrayKlass 对象进行表示），用于在 JVM 中表示一个 Java 类的对等体。当明确 OOP-Klass 模型的作用后，请大家思考一下 JVM 是如何通过栈帧中的对象引用访问到其内部的对象实例的呢？如图 6-5 所示，JVM 可以通过对象引用准确定位到 Java 堆区中的 instanceOopDesc 对象，这样既可成功访问到对象的实例信息，当需要访问目标对象的具体类型时，JVM 则会通过存储在 instanceOopDesc 中的元数据指针定位到存储在方法区中的 instanceKlass 对象上。

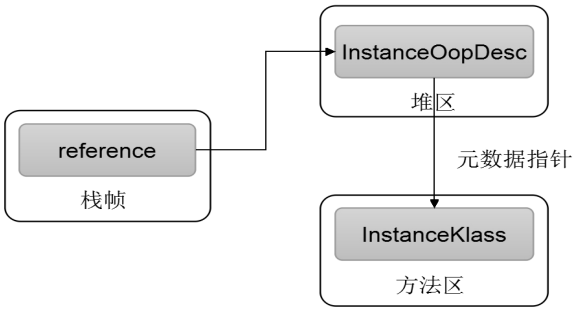


图 6-5　对象访问定位

6.5.4　GC的作用

GC（Garbage Collector，垃圾收集器）就是 JVM 中自动内存管理机制的具体实现。在 HotSpot 中，GC 的工作任务主要可以划分为两大块，分别是内存的动态分配和垃圾回收。而在执行内存分配之前，GC 首先会对内存空间进行划分，之前曾经提及过，考虑到 JVM 中存活对象的生命周期会具有两极化，因此应该采取不同的垃圾收集策略，分代收集由此诞生。目前几乎所有的 GC 都是采用分代收集算法执行垃圾回收的，所以 Java 堆区如果还要更进一步细分的话，还可以划分为新生代（YoungGen）和老年代（OldGen），其中新生代内又可以划分为 Eden 空间、From Survivor 空间和 To Survivor 空间，换句话说，内存空间究竟应该如何划分完全依赖于 GC 的设计。当内存空间划分完成后，GC 就可以为新对象分配内存空间，并区分出存储在内存中的对象哪些是存活的，哪些是已经死亡了的，如果对象已经死亡，那么就可以将其标记为垃圾。为了避免内存溢出，GC 就会释放掉无用对象所占用的内存空间，便于有足够的可用内存空间分配给新的对象实例。

大家思考一下，被标记为垃圾的无用对象所占用的内存空间何时进行回收？一般来说当内存空间中的内存消耗达到了一定阈值的时候，GC 就会执行垃圾回收，而且回收算法必须

非常精确，一定不能造成内存中存活的对象被错误回收掉，也不能造成已经死亡的对象没有被及时地回收掉。而且 GC 执行内存回收的时候应该做到高效，不应该导致应用程序出现长时间的暂停，以及避免产生内存碎片。不过当 GC 执行垃圾回收时，不可避免地会产生一些内存碎片，因为被回收的内存空间极有可能是一些不连续的内存块，这样一来将会导致没有足够的连续可用内存分配给较大的对象，不过可以使用压缩算法消除内存碎片（请阅读 6.5.6 节）。

在许多情况下，GC 不应该成为影响系统性能的瓶颈，所以参考《HotSpot 内存管理白皮书》的描述来看，可以根据以下 6 点来评估一款 GC 的性能，如下所示：

- 吞吐量：程序的运行时间/(程序的运行时间+内存回收的时间)；
- 垃圾收集开销：吞吐量的补数，垃圾收集器所占时间与总时间的比例；
- 暂停时间：执行垃圾收集时，程序的工作线程被暂停的时间；
- 收集频率：相对于应用程序的执行，收集操作发生的频率；
- 堆空间：Java 堆区所占的内存大小；
- 快速：一个对象从诞生到被回收所经历的时间。

6.5.5 垃圾标记：根搜索算法

在 GC 执行垃圾回收之前，首先需要区分出内存中哪些是存活对象，哪些是已经死亡的对象，只有被标记为已经死亡的对象，GC 才会在执行垃圾回收时，释放掉其所占用的内存空间，因此这个过程我们可以称之为垃圾标记阶段。那么在 JVM 中究竟是如何标记一个死亡对象呢？简单来说，当一个对象已经不再被任何的存活对象继续引用时，就可以宣判为已经死亡。

目前有两种比较常见的垃圾标记算法，分别是引用计数算法和根搜索算法。只不过在 HotSpot 和大部分 JVM 中则是使用后者作为垃圾标记的算法实现。尽管引用计数算法实现简单，执行效率也不错，但是该算法本身却存在一个较大的弊端，甚至会影响到垃圾标记的精准性。由于引用计数算法会为程序中的每一个对象都创建一个私有的引用计数器，当目标对象被其他存活对象引用时，引用计数器中的值则会加 1，不再引用时便会减 1，当引用计数器中的值为 0 的时候，就意味着该对象已经不再被任何存活对象引用，可以被标记为垃圾对象。采用这种方式看起来似乎没有任何问题，但是如果一些明显已经死亡了的对象尽管没有被任何的存活对象引用，但是它们彼此之间却存在相互引用时，引用计数器中的值则永远不会为 0，这样便会导致 GC 在执行内存回收时永远无法释放掉无用对象所占用的内存空间，极有可能引发内存泄露。

考虑到引用计数算法无法有效标记出已经死亡但存在相互引用的垃圾对象，JVM 的设

计者们则使用了另外一种垃圾标记算法标记垃圾对象，那就是根搜索算法。相对于引用计数算法而言，根搜索算法不仅同样具备实现简单和执行高效等特点，更重要的是该算法可以有效地解决在引用计数算法中一些已经死亡的对象因相互引用而导致的无法正确被标记的问题，防止内存泄露的发生。简单来说，根搜索算法是以根对象集合作为起始点，按照从上至下的方式搜索被根对象集合所连接的目标对象是否可达（使用根搜索算法后，内存中的存活对象都会被根对象集合直接或者间接连接着），如果目标对象不可达时，就意味着该对象已经死亡，便可以在 instanceOopDesc 的 Mark World 中将其标记为垃圾对象。如图 6-6 所示。

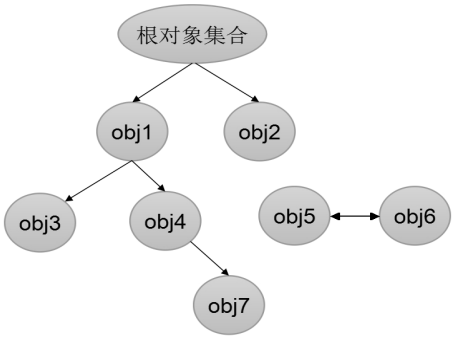


图 6-6　判断对象是否可达

在根搜索算法中，只有能够被根对象集合直接或者间接连接的对象才是存活对象，那么根对象集合中所存储的内容究竟是什么呢？在 HotSpot 中，根对象集合中包含了以下 5 个元素：

- Java 栈中的对象引用；
- 本地方法栈中的对象引用；
- 运行时常量池中的对象引用；
- 方法区中类静态属性的对象引用；
- 与一个类对应的唯一数据类型的 Class 对象。

6.5.6　垃圾回收：分代收集算法

6.5.5 节详细地讲解了关于垃圾标记的相关算法实现原理，那么当成功区分出内存中存活对象和死亡对象后，GC 接下来的任务就是执行垃圾回收释放掉无用对象所占用的内存空间，以便有足够的可用内存空间为新对象分配内存。由于执行垃圾回收的算法非常多，所以本书并不打算全部都一一列举，只会选取目前在 JVM 中比较常见的三种垃圾收集算法进行讲解，如下所示：

- 标记—清除算法（Mark-Sweep）；
- 复制算法（Copying）；

□ 标记—压缩算法（Mark-Compact）。

标记—清除算法是一种非常基础和常见的垃圾收集算法，它将垃圾回收任务划分为两个阶段执行，分别是垃圾标记和内存释放。相对于另外两种内存回收算法而言，标记—清除算法不仅执行效率低下，更重要的是，由于被执行内存回收的无用对象所占用的内存空间有可能是一些不连续的内存块，不可避免地会产生一些内存碎片，从而导致后续没有足够的可用内存空间分配给较大的对象。

为了提升GC在垃圾标记和内存释放这两个阶段的执行效率，JVM的设计者们提供了另外一种垃圾收集算法，那就是复制算法。由于JVM中的绝大多数对象都是瞬时状态，生命周期非常短暂，所以复制算法被广泛应用于新生代中。之前笔者曾经提及过，基于分代的概念，Java堆区如果还要更进一步细分的话，还可以划分为新生代（YoungGen）和老年代（OldGen），其中新生代又可以划分为Eden空间、From Survivor空间和To Survivor空间。在HotSpot中，Eden空间和另外两个Survivor空间缺省所占的比例是 8:1，当然开发人员可以通过选项“-XX:SurvivorRatio”调整这个空间比例。当执行一次Minor GC（新生代的垃圾回收）时，Eden空间中的存活对象会被复制到To空间内，并且之前已经经历过一次Minor GC并在From空间中存活下来的对象如果还年轻的话同样也会被复制到To空间内。在此大家需要注意，在满足两种特殊情况下，Eden和From空间中的存活对象将不会被复制到To空间内。首先是如果存活对象的分代年龄超过选项“-XX:MaxTenuringThreshold”所指定的阈值时，将会直接晋升到老年代中；其次当To空间的容量达到阈值时，存活对象同样也是直接晋升到老年代中，如图 6-7 所示[②]。 当所有的存活对象都被复制到To空间或者晋升到老年代后，剩下的均为垃圾对象，这就意味着GC可以对这些已经死亡了的对象执行一次Minor GC，释放掉其所占用的内存空间。

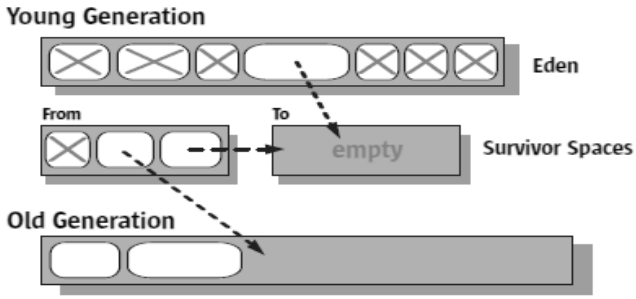


图 6-7 使用复制算法执行 Minor GC 之前

当执行完 Minor GC 之后，Eden 空间和 From 空间将会被清空，而存活下来的对象则会被全部存储在 To 空间内，接下来 From 空间和 To 空间将会互换位置，如图 6-8 所示。其实

② 图 6-7、图 6-8、图 6-10、图 6-11、图 6-12 均来源于《HotSpot 内存管理白皮书》中。

复制算法无非就是使用 To Survivor 空间作为一个临时的空间交换角色，务必需要保证两块 Survivor 空间中一块必须是空的，这就是复制算法。尽管复制算法能够高效执行 Minor GC，但是它却并不适用于老年代中的内存回收，因为老年代中对象的生命周期都比较长，甚至在某些极端的情况下还能够与 JVM 的生命周期保持一致，所以如果老年代也采用复制算法执行内存回收不仅需要额外的时间和空间，而且还会导致较多的复制操作影响到 GC 的执行效率。

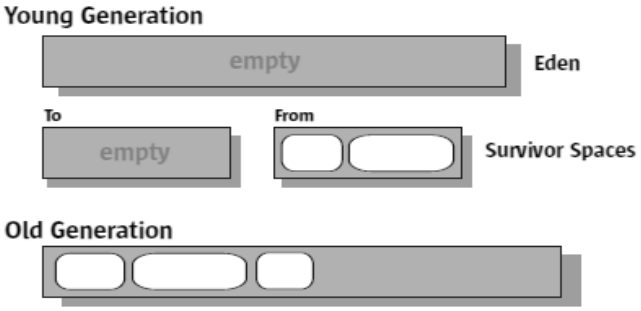


图 6-8　使用复制算法执行 Minor GC 之后

标记—清除算法的确可以应用在老年代中，但是该算法不仅执行效率低下，而且在执行完内存回收后还会产生内存碎片，所以 JVM 的设计者们在此基础之上进行了改进，标记—压缩算法由此诞生。当成功标记出内存中的垃圾对象后，该算法会将所有的存活对象都移动到一个规整且连续的内存空间中，然后执行 Full GC（老年代的垃圾回收，或者称之为 Major GC）回收无用对象所占用的内存空间。当成功执行压缩之后，已用和未用的内存都各自一边，彼此之间维系着一个记录下一次分配起始点的标记指针，当为新对象分配内存时，则可以使用指针碰撞（Bump the Pointer）技术修改指针的偏移量将新对象分配在第一个空闲内存位置上，为新对象分配内存带来便捷。

在 HotSpot 中，基于分代的概念，GC 所使用的内存回收算法必须结合新生代和老年代等各自的特点，简单来说，就是针对不同的代空间，从而组合使用不同的垃圾收集算法。参考《HotSpot 内存管理白皮书》，为新生代选择的垃圾收集算法通常是以速度优先，因为新生代中所存储的瞬时对象声明周期非常短暂，可以有针对性地使用复制算法，因此执行 Minor GC 时，一定要保持高效和快速。而新生代中的内存空间通常都比较小，所以回收新生代时一定会非常频繁。但老年代通常使用更节省内存的回收算法，因为老年代中所存储的对象声明周期都非常长，并且老年代占据了大部分的堆空间，所以老年代的 Full GC 并不会跟新生代的 Minor GC 一样频繁，不过一旦程序中发生一次 Full GC 时，将会耗费更长的时间来完成，那么在老年代中使用标记—清除算法或者标记—压缩算法执行垃圾回收将会是不错的选择。

6.6 垃圾收集器

当大家清楚 JVM 的内存回收算法后，接下来我们正式走进 JVM 自动内存管理的具体实现，即 GC。由于 JDK 的版本处于高速迭代中，因此 Java 从发展至今衍生了许多的 GC 版本，比如：Serial/Serial Old 收集器、Parallel/Parallel Old 收集器、CMS（Concurrent-Mark-Sweep）收集器，以及从 JDK7 Update4 版本开始提供的 G1（Garbage-First）收集器等。如图 6-9 所示。

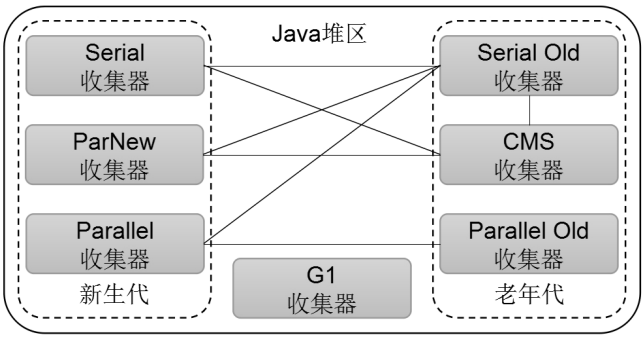


图 6-9 HotSpot 中的 GC

基于分代的概念，不同的代空间中均活动着不同的 GC，比如 Serial 收集器就是一个典型的新生代垃圾收集器，它采用复制算法回收新生代无用对象的内存空间。当然 JVM 在实际运行过程中，新生代和老年代中各自的 GC 需要组合在一起共同执行垃圾回收任务。如果新生代的 GC 和老年代的 GC 相连，则意味着可以组合在一起使用，不过在实际开发过程中，新生代和老年代的 GC 究竟应该如何组合，还需要结合具体的应用场景。

在正式开始讲解垃圾收集器之前，大家首先需要了解两个非常重要的概念，如下所示：

- 串行还是并行回收；
- 并发还是“Stop-the-World”机制。

串行回收指的就是在同一时间段内只允许一件事情发生，简单来说，当多个 CPU 可用时，也只能有一个 CPU 用于执行垃圾回收操作，并且在执行垃圾回收时，程序中的工作线程将会被暂停，当垃圾收集工作完成后才会恢复之前被暂停的工作线程，这就是串行回收。不过由于运行在客户端下的 Client 模式的 JVM 并没有低暂停时间的要求，并且 Client 模式下的内存开销相对于 Server 模式来说也更小，因此串行回收缺省被应用在 Client 模式下的 JVM 中。和串行回收相反，并行收集可以运用多个 CPU 同时执行垃圾回收，因此提升了应用程序的吞吐量，不过并行回收仍然使用了“Stop-the-World”机制和复制算法。如图 6-10 所示。

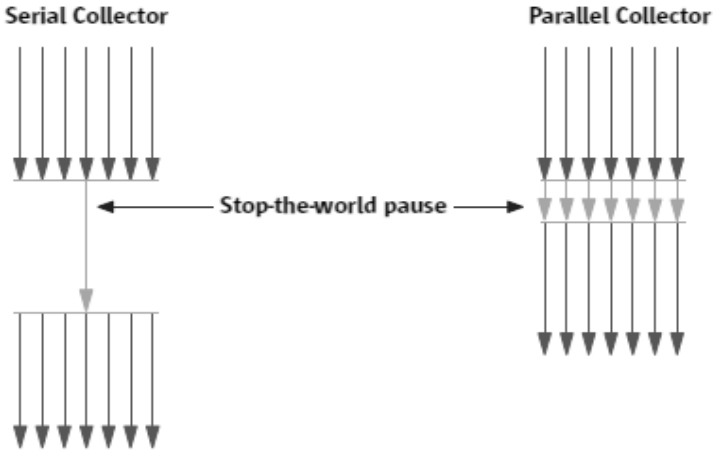


图 6-10 串行回收与并行回收的区别

探讨了串行回收和并行回收之后，接下来我们再来了解并发回收和“Stop-the-World”机制。当通过“Stop-the-World”机制的方式来运行垃圾收集器时，垃圾收集器会在内存回收的过程中暂停程序中所有的工作线程，直至完成内存回收后才会恢复之前被暂停的工作线程。如果“Stop-the-World”出现在新生代的 Minor GC 中时，由于新生代的内存空间通常都比较小，所以暂停时间也在可接受的合理范围之内，不过一旦出现在老年代的 Full GC 中时，程序的工作线程被暂停的时间将会更久，这往往直接跟 Java 堆空间所管理的内存大小有关。简单来说，内存空间越大，执行 Full GC 的时间就会越久，相对的工作线程被暂停的时间也就会更长。以互联网项目为例，由于项目的特殊性，因此经常需要用到多达几十乃至上百 GB 的内存，如果用户的登录操作恰巧碰见垃圾收集器在执行 Full GC 时，这将会是一场灾难，如果用户等待的时间过长，这就不是用户体验下降这么简单的事情了，甚至有可能会流失用户，所以 JVM 的设计者们提供了并发回收希望以此缩短“Stop-the-World”机制的暂停时间，一旦使用并发回收后，在同一时间段内，应用程序的工作线程和垃圾收集线程将会同时运行或交叉运行。在此大家需要注意，直至目前为止，哪怕是目前最新的 G1 收集器也做不到完全不需要“Stop-the-World”，所以说尽管如今垃圾收集器越来越优秀，回收效率也越来越高效，也只是尽可能地缩短暂停时间而已。

6.6.1 串行回收：Serial收集器

Serial 收集器作用于新生代中，它采用复制算法、串行回收和“Stop-the-World”机制的方式执行内存回收。在早期的 JDK 版本中，由于那个年代的硬件发展速度并没有像今天一样呈现井喷状态，所以在 CPU 受限于单个的情况下，使用 Serial 收集器执行新生代垃圾收集几乎是唯一的选择，并且 Serial 收集器缺省也作为 HotSpot 中 Client 模式下的新生代垃圾收集器。

除了新生代之外，Serial 收集器还提供用于执行老年代垃圾收集的 Serial Old 收集器。Serial Old 收集器同样也采用了串行回收和“Stop-the-World”机制，只不过内存回收算法使用的却是标记—压缩算法。在此大家需要注意，如果 JVM 受限在单个 CPU 的宿主环境中，使用 Serial 收集器+Serial Old 收集器的组合执行 Client 模式下的内存回收将会是不错的选择。之前曾经提及过，基于串行回收的垃圾收集器适用于大多数对暂停时间要求不高的 Client 模式下的 JVM 中，由于 CPU 不需要频繁地做任务切换，因此可以有效避免多线程交互过程中产生的一些额外开销，虽然执行串行回收会降低程序的吞吐量，但是回收质量还是不错的。在程序中，开发人员可以通过选项“-XX:+UseSerialGC”手动指定使用 Serial 收集器执行内存回收任务。

6.6.2 并行回收：ParNew收集器

如果说 Serial 是新生代中的单线程垃圾收集器，那么 ParNew 收集器则是 Serial 收集器的多线程版本。ParNew 收集器除了采用并行回收的方式执行内存回收外，两款垃圾收集器之间几乎没有任何区别，因为 ParNew 收集器在新生代中同样也是采用复制算法和“Stop-the-World”机制。

大家思考一下，由于 ParNew 收集器是基于并行回收，那么是否可以断定 ParNew 收集器的回收效率在任何场景下都会比 Serial 收集器更高效？如果说 ParNew 收集器运行在多 CPU 的宿主环境下，由于可以充分利用多 CPU、多核心等物理硬件资源优势，确实可以更迅速地完成垃圾收集，提升程序的吞吐量。但是如果是在 CPU 受限于单个的情况下，ParNew 收集器不见得会比 Serial 收集器更高效，虽然 Serial 收集器是基于串行回收，但由于 CPU 不需要频繁地做任务切换，因此可以有效避免多线程交互过程中产生的一些额外开销。所以从理论上来说，Serial 收集器的优势是在 JVM 受限于单 CPU 的宿主环境中，而 ParNew 收集器的优势则是体现在多 CPU、多核心等宿主环境中，并且在某些注重低延迟的应用场景中，ParNew 收集器+CMS（Concurrent-Mark-Sweep）收集器组合执行 Server 模式下的内存回收几乎是最佳选择。在程序中，开发人员可以通过选项“-XX:+UseParNewGC”手动指定使用 ParNew 收集器执行内存回收任务。

6.6.3 程序吞吐量优先：Parallel收集器

就目前而言，HotSpot 的新生代中除了拥有 ParNew 收集器是基于并行回收的外，Parallel 收集器同样也采用了复制算法、并行回收和“Stop-the-World”机制。不过由于 ParNew 收集器已经采用并行回收，在多 CPU 的宿主环境中必然能够提升 GC 的回收效率和程序的吞吐量，那么 Parallel 收集器的出现是否多此一举？和 ParNew 收集器不同，Parallel 收集器可以控制程序的吞吐量大小，因此它也被称作吞吐量优先的垃圾收集器。在程序中，开发人员可

以通过选项“-XX:GCTimeRatio”设置执行内存回收的时间所占 JVM 运行总时间的比例，也就是控制 GC 的执行频率，公式为 1/(1+*N*)，默认值为 99，也就是说，将只有 1%的时间用于执行内存回收。除此之外，Parallel 收集器还提供选项“-XX:MaxGCPauseMillis”设置执行内存回收时“Stop-the-World”机制的暂停时间阈值，如果指定了该选项，那么 Parallel 收集器将会尽可能地在设定的时间范围内完成内存回收。

在此大家需要注意，在垃圾收集器中吞吐量和低延迟这两个目标其实是存在相互竞争的矛盾，因为如果选择以吞吐量优先，那么降低内存回收的执行频率则是必然的，但这将会导致 GC 需要更长的暂停时间来执行内存回收。相反如果选择以低延迟优先，那么为了降低每次执行内存回收时的暂停时间，只能够频繁地执行内存回收，但这又引起了新生代内存的缩减和导致程序吞吐量的下降。举个例子，在 60s 的 JVM 总运行时间里，每次 GC 的执行频率是 20s/次，那么 60s 内一共会执行 3 次内存回收，按照每次 GC 耗时 100ms 来计算，最终一共会有 300ms（即 60/20*100）被用于执行内存回收。但是如果我们将选项“-XX:MaxGCPauseMillis”的值调小后，新生代的内存空间也会自动调整，相信大家都知道，内存空间越小就越容易被耗尽，那么 GC 的执行频率就会更频繁。之前在 60s 的 JVM 总运行时间里，最终会有 300ms 被用于执行内存回收，而如今 GC 的执行频率却是 10s/次，60s 内将会执行 6 次内存回收，按照每次 GC 耗时 80ms 来计算，虽然看上去暂停时间更短了，但最终一共会有 480ms（即 60/10*80）被用于执行内存回收，很明显程序的吞吐量下降了。所以大家在设置这两个选项时，一定需要注意控制在一个折中的范围之内。Parallel 收集器还提供一个“-XX:UseAdaptiveSizePolicy”选项用于设置 GC 的自动分代大小调节策略，一旦设置这个选项后，就意味着开发人员将不再需要显示的设置新生代中的一些细节参数了，JVM 会根据自身的当前运行情况动态调整这些相关参数。

和 Serial 收集器一样，Parallel 收集器也提供用于执行老年代垃圾收集的 Parallel Old 收集器，Parallel Old 收集器采用了标记—压缩算法，但同样也是基于并行回收和“Stop-the-World”机制。在程序吞吐量优先的应用场景中，Parallel 收集器+Parallel Old 收集器组合执行 Server 模式下的内存回收将会是不错的选择。在程序中，开发人员可以通过选项“-XX:+UseParallelGC”手动指定使用 Parallel 收集器执行内存回收任务。

6.6.4 低延迟：CMS（Concurrent-Mark-Sweep）收集器

尽管在程序吞吐量优先的应用场景中，Parallel 收集器+Parallel Old 收集器组合执行 Server 模式下的内存回收将会是不错的选择，但是在某些对系统响应速度要求比较高的项目中，大家总是希望系统能够快速做出响应，而不愿意看到过多的延迟。基于低延迟的考虑，JVM 的设计者们提供了基于并行回收的 CMS（Concurrent-Marking-Sweep）收集器，它是一款优秀的老年代垃圾收集器，也可以称作 Mostly-Concurrent 收集器。CMS 天生为并发而生，

低延迟是它的优势，不过垃圾收集算法却并没有采用标记—压缩算法，而是采用标记—清除算法，并且也会因为“Stop-the-World”机制而出现短暂的暂停。在此大家需要注意，和之前本书介绍过的所有垃圾收集器都不同，CMS 的实现细节更加复杂，其执行过程主要可以划分为 4 个阶段，如下所示：

- 初始标记（Initial-Mark）阶段；
- 并发标记（Concurrent-Marking）阶段；
- 再次标记（Remark）阶段；
- 并发清除（Concurrent-Sweep）阶段。

CMS 收集器的回收周期以一个称之为初始标记的阶段开始，在这个阶段中，程序中所有的工作线程都将会因为“Stop-the-World”机制而出现短暂的暂停，这个阶段的主要任务就是标记出内存中那些被根对象集合所连接的目标对象是否可达，一旦标记完成之后就会恢复之前被暂停的所有应用线程。接下来将会进入并发标记阶段，而这个阶段的主要任务就是将之前的不可达对象标记为垃圾对象。在 CMS 最终执行内存回收之前，尽管看上去这些垃圾对象都已经被成功标记了，但是由于在并发标记阶段中，程序的工作线程会和垃圾收集线程同时运行或者交叉运行，因此在并发标记阶段将无法有效确保之前被标记为垃圾的无用对象的引用关系遭到更改，为了解决这个问题，CMS 会进入到再次标记阶段，这样一来，程序会因为“Stop-the-World”机制而再次出现短暂的暂停，以确保这些垃圾对象都能够被成功且正确地标记。当经历过初始标记、并发标记和再次标记三个阶段后，CMS 最终将会进入到并发清除阶段执行内存回收，释放掉无用对象所占用的内存空间。如图 6-11 所示。

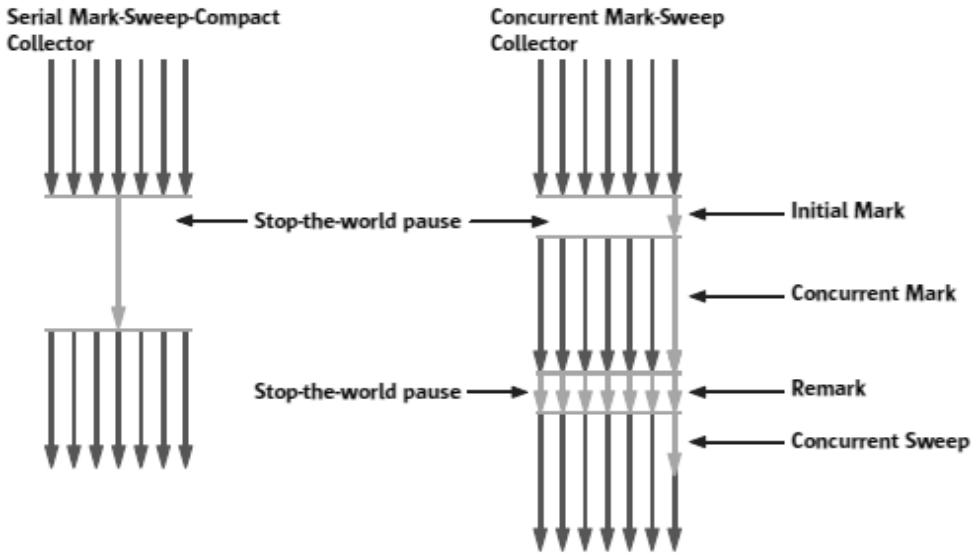


图 6-11 基于并发回收的 CMS 收集器

尽管 CMS 收集器采用的是并行回收，但是在其初始标记和再次标记这两个阶段中仍然需要执行“Stop-the-World”机制暂停程序中的工作线程，不过暂停时间并不会太长，因此可以说明目前所有的垃圾收集器都做不到完全不需要“Stop-the-World”，只是尽可能地降低暂停时间而已。

在之前笔者介绍过的几款老年代垃圾收集器中，比如：Serial Old 收集器、Parallel Old 收集器的垃圾收集算法均是采用标记—压缩来避免执行 Full GC 后产生内存碎片，而 CMS 收集器的垃圾收集算法采用的却是标记—清除，这意味着每次执行完内存回收后，由于被执行内存回收的无用对象所占用的内存空间极有可能是不连续的一些内存块，不可避免地将会产生一些内存碎片，如图 6-12 所示。那么 CMS 在为新对象分配内存空间时，将无法使用指针碰撞（Bump the Pointer）技术，而只能够选择空闲列表（Free List）执行内存分配。

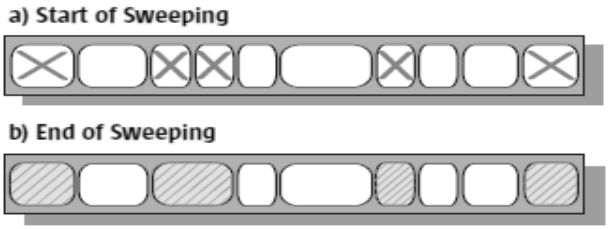


图 6-12　标记—清除之后产生的内存碎片

在 HotSpot 中，当垃圾收集器执行完内存回收后，如果内存空间中产生内存碎片，那么则只能够选择空闲列表作为内存分配算法为新对象分配内存空间。简单来说，会有 JVM 负责维护一个列表，其中所记录的内容就是当前内存空间中可用内存块的坐标，当执行内存分配时，会从列表中定位到一个与新对象所需内存大小一致的连续内存块用于存储生成的对象实例。考虑到内存碎片存在的弊端，CMS 收集器提供选项“-XX:+UseCMS-CompactAtFullCollection”，用于指定在执行完 Full GC 后是否对内存空间进行压缩整理，以此避免内存碎片的产生。不过由于内存压缩整理过程无法并发执行，所带来的问题就是停顿时间将会变得更加长，因此 CMS 收集器还提供另外一个选项“-XX:CMSFullGCs-BeforeCompaction”，用于设置在执行多少次 Full GC 后对内存空间进行压缩整理。除了会产生内存碎片外，CMS 收集器还存在一个不容忽视的问题，那就是在并发标记阶段由于程序的工作线程和垃圾收集线程是同时运行或者交叉运行的，那么在并发标记阶段如果产生新的垃圾对象时，CMS 将无法对这些垃圾对象进行标记，最终会导致这些新产生的垃圾对象没有被及时地回收掉，从而只能够在下一次执行 GC 时，释放掉这些之前未被回收的内存空间。

在此大家需要注意，尽管 Full GC 几乎已经成为老年代垃圾回收的代名词，但实际上 Full GC 的回收范围却不单单仅限于老年代中，从严格意义上来说，Full GC 的回收范围几

乎覆盖了整个堆空间，因此 Full GC 将会比 Minor GC 耗费更长的时间来完成垃圾收集。在 HotSpot 中，除了 CMS 收集器之外的任何其他老年代垃圾收集器在执行内存回收时，都将会执行 Full GC（G1 收集器的垃圾收集动作比较特殊，被称之为 Mixed GC），而 CMS 收集器却提供选项“-XX:CMSInitiatingOccupancyFraction”用于设置当老年代中的内存使用率达到多少百分比的时候执行内存回收（低版本的 JDK 缺省值为 68%，JDK6 及以上版本默认值则为 92%），这里的内存回收范围仅限于老年代，而非整个堆空间，因此通过该选项便可以有效降低 Full GC 的执行次数。当然并不是说使用了 CMS 收集器之后，就永远不会再触发 Full GC 了，一旦 CMS 在执行过程中出现“Promotion Failed”或“Concurrent Mode Failure”时，将仍然有可能会触发 Full GC 操作。在程序中，开发人员可以通过选项“-XX:+UseConcMarkSweepGC”来手动指定使用 CMS 收集器执行内存回收任务。

6.6.5 区域化分代式：G1（Garbage-First）收集器

G1 收集器的设计初衷是为了替代 CMS 收集器而生，简单来说，G1 是一款基于并行和并发、低延迟以及暂停时间更加可控的区域化分代式垃圾收集器。和之前本书介绍过的所有垃圾收集器都不同（包括 CMS 收集器），G1 在设计上的改变可以说是具有革命性的意义。首先在内存空间的设计上，G1 重新塑造了整个 Java 堆区，尽管同样也是基于分代的概念执行内存分配和垃圾回收，但是 G1 却并没有采用传统物理隔离的新生代和老年代布局方式（仅逻辑上划分为新生代和老年代），而是选择将 Java 堆区划分成约 2 048 个大小相同的独立 Region 块，每个 Region 块之间可能是不连续的，其大小根据堆空间的实际大小而定，整体被控制在 1MB 到 32MB 之间，如图 6-13 所示。这样划分的好处在于可以更好地提升 GC 的回收效率和缩短“Stop-the-World”机制的暂停时间以换取更大的程序吞吐量，甚至能够真正做到精准控制程序的暂停时间等。这是因为 G1 收集器在执行内存回收时，会优先释放掉整个 Java 堆区中一些占用内存较大的 Region 块，从而可以避免像以往一样直接扫描整个 Java 堆区（如果一个 Region 块中引用另一个 Region 块内的对象时，则通过 Remembered Set 技术避免全堆扫描）。至于执行内存回收时“Stop-the-World”机制的暂停时间更加可控，是相对于 Parallel 收集器而言，尽管 Parallel 收集器也提供选项“-XX:MaxGCPauseMillis”控制程序的暂停时间，但是在大内存的空间释放操作中，暂停时间其实并非是完全可控的，只能说是尽可能控制在预期范围内，但谁也无法保证内存回收时所带来的暂停时间一定会百分百地控制在规定的时间阈值内，所以必然会有一定的误差。而由于 G1 收集器并非全堆扫描，只优先回收占用内存较大的一些 Region 块，所以 G1 收集器的暂停时间会更加可控。

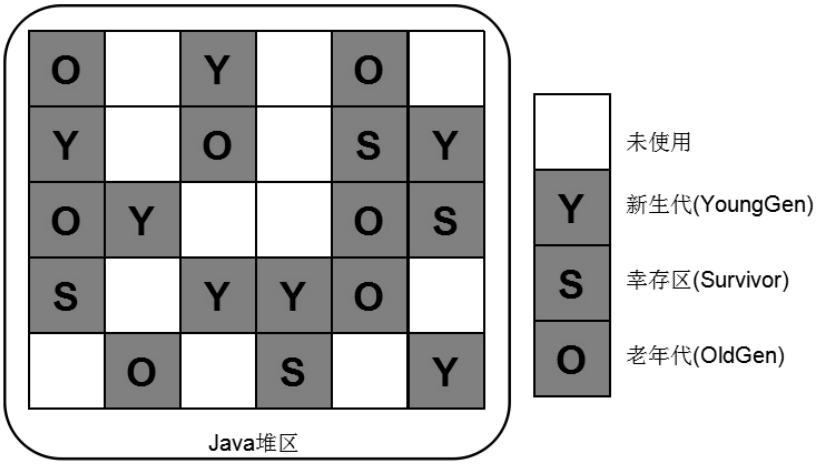


图 6-13 基于 Region 块的 Java 堆区布局

G1 收集器的执行过程主要可以划分为 6 个阶段，如下所示：

- 初始标记（Initial-Mark）阶段；
- 根区域扫描（Root-Region-Scanning）阶段；
- 并发标记（Concurrent-Marking）阶段；
- 再次标记（Remark）阶段；
- 清除（Cleanup）阶段；
- 拷贝（Copying）阶段。

G1 收集器的执行过程与 CMS 收集器相似。在初始标记阶段中，程序中所有的工作线程都将会因为“Stop-the-World”机制而出现短暂的暂停，该阶段的主要任务是标记 Root-Region。一旦标记完成之后就会恢复之前被暂停的所有应用线程。接下来将会进入到根区域扫描阶段，该阶段的主要任务是扫描 Root-Region 中引用老年代的一些 Region 块，只有在执行完该阶段之后，才能开始下一次新生代内存回收。

并发标记阶段的主要任务就是找出整个Java堆区中的存活对象，由于该阶段并不会导致程序出现暂停，因此在执行的过程中允许被新生代内存回收打断。再次标记阶段和初始标记阶段同样都是基于“Stop-the-World”机制的，该阶段的主要任务就是完成整个堆区中存活对象的标记。清除标记阶段由三部分构成，首先会计算出所有的活跃对象并完全释放一些自由的Region块，然后处理Remembered Set，在此大家需要注意，这两部分操作将会暂停程序中的应用线程，然后并发重置空闲的一些Region块，并将它们放回至空闲列表中。最后的拷贝阶段也是基于“Stop-the-World”机制的，该阶段的主要任务是将存活对象复制到未使用过的Region块中。当经历过这 6 个阶段后，G1 收集器的内存回收任务就算是完成了（关于

G1 收集器的更多介绍请参考Oracle的官方文档[③]）。在程序中，开发人员可以通过选项“-XX:+UseG1GC”来手动指定使用G1 收集器执行内存回收任务。

在此大家需要注意，尽管 G1 收集器长期以来的目标是为了逐步替代掉 CMS 收集器，但这并非是要求大家立刻将生产环境中 JDK 版本升级到 JDK7 Update4 及更高的版本，或者将 CMS 收集器切换到 G1 收集器，如果目前程序中并没有因为“Stop-the-World”机制导致程序出现长时间的暂停，应该尽可能保持当前现状不变，避免因此导致程序出现一些无法预估的风险。

6.6.6 垃圾收集的相关选项配置

对于一名专业的 JVM 调优人员而言，务必需要非常熟悉虚拟机的各项常用选项配置，因为只有这样才能够更好地对 JVM 进行调优和调试。总体来说，JVM 的选项配置大致可以划分为标准选项和非标准选项，其中标准选项是 Java 虚拟机规范强制要求各类 JVM 必须实现的，而非标准选项则相反，因为非标准选项极有可能在后续的迭代版本中被更改或者取消掉。

表 6-1 内存选项配置

选项	缺省值	描述	备注
-Xms	物理内存的 1/64(<1GB)	设置 Java 堆区的初始内存	当可用的 Java 堆区内存小于 40%时，JVM 就会将内存调整到选项-Xmx 所允许的最大值
-Xmx	物理内存的 1/4(<1GB)	设置 Java 堆区的最大内存	当可用的 Java 堆区内存大于 70%时，JVM 就会将内存调整到选项-Xms 所指定的初始值
-Xmn	暂无	设置新生代(YoungGen)的内存	-Xmn 的内存大小为 Eden 空间+2 个 Survivor 空间的值，官方建议配置为整个堆的 3/8
-XX:NewSize	暂无	设置新生代(YoungGen)的初值内存	和选项-Xmn 等价，但推荐生产环境中使用-Xmn，相当于一次性设定了 NewSize/Max-NewSIze 的内存大小
-XX:MaxNewSize	暂无	设置新生代(YoungGen)的最大内存	
-XX:NewRatio	暂无	新生代（Eden 空间+2 个 Survivor 空间）与老年代的比值，不包括永久代	选项-XX:NewRatio=4 时，表示新生代与老年代所占的比值为 1:4。如果已经设置了选项 Xmn，则无须设置该选项
-XX:PermSize	物理内存的 1/64	设置方法区的初始内存	暂无
-XX:MaxPermSize	物理内存的 1/4	设置方法区的最大内存	暂无

③ G1 收集器调优：http://www.oracle.com/technetwork/cn/articles/java/g1gc-1984535-zhs.html。

续表

选项	缺省值	描述	备注
-XX:SurvivorRatio	暂无	Eden 空间与 2 个 Survivor 空间的比值大小	Eden 空间和另外 2 个 Survivor 空间缺省所占的比例是 8:1
-XX:TLABWasteTargetPercent	1%	设置 TLAB 空间所占用 Eden 空间的百分比大小	暂无

表 6-2 GC 组合配置

GC 组合	Minor GC	Full GC	描述
-XX:+UseSerialGC	Serial 收集器串行回收	Serial Old 收集器串行回收	选项-XX:+UseSerialGC 可以手动指定使用 Serial 收集器+Serial Old 收集器组合执行内存回收
-XX:+UseParNewGC	ParNew 收集器并行回收	Serial Old 收集器串行回收	选项-XX:+UseParNewGC 可以手动指定使用 ParNew 收集器+Serial Old 收集器组合执行内存回收
-XX:+UseParallelGC	Parallel 收集器并行回收	Serial Old 收集器串行回收	通过-XX:+UseParallelGC 可以手动指定使用 Parallel 收集器+Serial Old 收集器组合执行内存回收
-XX:+UseParallelOldGC	Parallel 收集器并行回收	Parallel Old 收集器并行回收	通过-XX:+UseParallelOldGC 可以手动指定使用 Parallel 收集器+Parallel Old 收集器组合执行内存回收
-XX:+UseConcMarkSweepGC	ParNew 收集器并行回收	缺省使用 CMS 收集器并发回收，备用采用 Serial Old 收集器串行回收	使用-XX:+UseConcMarkSweepGC 可以手动指定使用 ParNew 收集器+CMS 收集器+Serial Old 收集器组合执行内存回收。优先使用 ParNew+CMS 组合，当出现 ConcurrentMode Failure 或 Promotion Failed 时，则采用 ParNew+Serial Old 组合
-XX:+UseConcMarkSweepGC -XX:-UseParNewGC	Serial 收集器串行回收		
-XX:+UseG1GC	G1 收集器并发、并行执行内存回收		暂无

表 6-3 GC 选项配置

GC 名称	选项	描述
Serial 收集器	-XX:MaxTenuringThreshold	GC 分代年龄设置，缺省新生代(YoungGen)存活对象晋升老年代(OldGen)的年龄是 15 岁
ParNew 收集器	-XX:MaxTenuringThreshold	GC 分代年龄设置，缺省新生代(YoungGen)存活对象晋升老年代(OldGen)的年龄是 15 岁
	-XX:+UseAdaptiveSizePolicy	动态调整 Java 堆区内各个区域的内存大小，以及 GC 分代年龄
Parallel/Parallel Old 收集器	-XX:ParallelGCThreads	垃圾收集的线程数
	-XX:+UseAdaptiveSizePolicy	动态调整 Java 堆区内各个区域的内存大小，以及 GC 分代年龄
	-XX:MaxTenuringThreshold	GC 分代年龄设置，缺省新生代(YoungGen)存活对象晋升老年代(OldGen)的年龄是 15 岁

续表

GC 名称	选项	描述
Parallel/Parallel Old 收集器	-XX:GCTimeRatio	设置执行内存回收的时间所占 JVM 运行总时间的比例(缺省值为 99)，公式为 1/(1+*N*)
	-XX:MaxGCPauseMillis	设置执行内存回收时“Stop-the-World”机制的暂停时间阈值(该选项仅对 Parallel 收集器有效)
CMS 收集器	-XX:ParallelCMSThreads	垃圾收集的线程数
	-XX:CMSFullGCsBeforeCompaction	由于 CMS 采用标记—清除算法，所以在执行完内存回收之后会产生内存碎片。该选项用于设置执行多少次 Full GC 后对内存空间进行压缩整理
	-XX:+UseCMSCompactAtFullCollection	由于 CMS 采用标记—清除算法，所以在执行完内存回收之后会产生内存碎片。该选项用于设置当执行 Full GC 后是否对内存空间进行压缩整理
CMS 收集器	-XX:CMSInitiatingOccupancyFraction	当老年代中的内存使用率达到多少百分比的时候执行内存回收（低版本的 JDK 缺省值为 68%，JDK6 及以上版本缺省值则为 92%）
	-XX:+UseCMSInitiatingOccupancyOnly	允许 HotSpot 根据成本来决定什么时候执行内存回收（缺省为 false）
	-XX:+CMSClassUnloadingEnabled	方法区采用 CMS 收集器执行内存回收
	-XX:+CMSIncrementalMode	在单个 CPU 的情况下，使用增量模式
	XX:CMSInitiatingPermOccupancyFraction	设置方法区的内存使用率到达阈值后执行内存回收
G1 收集器	-XX:G1ReservePercent	设置作为空闲空间的预留内存百分比，以降低目标空间溢出的风险（缺省值为 10%）
	-XX:G1HeapRegionSize	设置 Region 的大小，值为 2 的幂，范围为 1 MB 到 32 MB，目标是根据最小的 Java 堆内存大小划分出约 2048 个 Regions

表 6-4 GC 辅助选项配置

选项	描述
-XX:+PrintGC	打印垃圾收集器的基础信息
-XX:+PrintGCDetails	打印垃圾收集器更详细的信息
-XX:+PrintGCTimeStamps	打印垃圾收集器开始执行的时间戳（基准时间）
-XX:+PrintGCDateStamps	打印垃圾收集器开始执行的时间戳（日期格式）
-XX:+PrintGCApplicationStoppedTime	打印垃圾收集器造成程序暂停的时间
-XX:+PrintTLAB	打印 TLAB 空间的使用情况
-XX:+PrintTenuringDistribution	打印每次 Minor GC 后新的存活周期的阈值
-XX:+PrintHeapAtGC	打印 GC 前后详细的堆栈信息
-Xloggc	指定 GC 日志的输出路径

6.7 实战：GC日志分析

之前曾经提及过，尽管 JVM 的自动内存管理机制大大提升了 Java 开发人员的编程效率，甚至从某种意义上来说还降低了内存泄露和内存溢出的风险，但是如果 Java 开发人员过度依赖于“自动”，那么这将会是一场灾难，最严重的就是会弱化 Java 开发人员在程序出现内存溢出时定位问题和解决问题的能力。除此之外，当 GC 沦落为影响程序运行的性能瓶颈时，比如因为“Stop-the-World”机制导致程序出现长时间的暂停，或者 GC 频繁执行内存回收导致程序吞吐量下降等情况，我们必然需要通过分析这些 GC 日志来做出相应的调整和处理。也就是说，分析 GC 日志是故障排查和 JVM 性能调优的前提和基础，当然本章的内容并不会涉及具体的调优细节，仅仅只会对 GC 日志做离线分析，让大家理解 GC 日志中的内容，尽可能避免因看不懂 GC 日志而在故障排查和性能调优上无从下手。

6.7.1 不同GC日志的展示形式

在 HotSpot 中，Serial/Serial Old 收集器、ParNew 收集器、Parallel/Parallel Old 收集器、CMS 收集器和 G1 收集器等每一款垃圾收集器都各自拥有特定的日志展示形式。不过大家不必担心，理解每一种垃圾收集器打印的 GC 日志其实并不困难，因为除了 CMS 收集器和 G1 收集器打印的 GC 日志稍显特殊外，其他的在展示形式上其实都大同小异。关于如何在程序运行过程中打印出 GC 日志，则可以参考表 6-4 中的选项来进行设置。

1．Serial/Serial Old 收集器的日志形式：

```
  [GC [DefNew: 21463K->375K(27648K), 0.0213175 secs] 21463K->20855K(99328K),
0.0213743 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
  [Full GC [Tenured: 375K->362K(71680K), 0.0043782 secs] 375K->362K(99328K),
[Perm : 148K->148K(12288K)], 0.0044761 secs] [Times: user=0.00 sys=0.00,
real=0.00 secs]
```

2．ParNew 收集器的日志形式：

```
  [GC [ParNew: 21463K->464K(27648K), 0.0450565 secs] 21463K->20944K(99328K),
0.0451287 secs] [Times: user=0.03 sys=0.02, real=0.05 secs]
```

3．Parallel/Parallel Old 收集器的日志形式：

```
  [GC-- [PSYoungGen: 21409K->21409K(26880K)] 82849K->82849K(98560K),
0.0104002 secs] [Times: user=0.05 sys=0.00, real=0.01 secs]
  [Full GC [PSYoungGen: 21409K->0K(26880K)] [ParOldGen: 61440K->41411K(71680K)]
82849K->41411K(98560K) [PSPermGen: 1616K->1614K(12288K)], 0.0467510 secs]
[Times: user=0.11 sys=0.00, real=0.05 secs]
```

接下来笔者首先对 Parallel/Parallel Old 收集器打印的 GC 日志为例，为大家具体讲解这

两款垃圾收集器打印的 GC 日志内容。仔细观察上述 GC 日志，程序在运行过程中 Minor GC 和 Full GC 各执行了一次。其中“[PSYoungGen: 21409K->21409K(26880K)]”代表执行完 Minor GC 后新生代的内存使用率，“PSYoungGen”为所回收的内存区类型，这与所使用的垃圾收集器是密切相关的（以 Parallel/Parallel Old 收集器为例，新生代为“PSYoungGen”、老年代为“ParOldGen”、永久代为“PSPermGen”），“21409K”为新生代回收前的内存使用率，“21409K”为新生代回收后的内存使用率，“26880K”为新生代的总内存大小。“82849K->82849K(98560K)”代表着 Java 堆区的整体内存使用率，“82849K”为堆区回收前的内存使用率，“82849K”为堆区回收后的内存使用率，“98560K”为 Java 堆区的总内存大小。“0.0104002 secs”代表着本次执行 Minor GC 的耗时，以秒为单位。“[Times: user=0.11 sys=0.00, real=0.05 secs]”中的 3 个参数分别为程序耗时、系统耗时和实际耗时。

当明白 Parallel 收集器打印的 GC 日志内容后，接下来我们再来分析 Parallel Old 收集器打印的 GC 日志。其中“[PSYoungGen: 21409K->0K(26880K)]”代表着新生代的内存使用率。“[ParOldGen: 61440K->41411K(71680K)]”代表执行完 Full GC 后老年代的内存使用率，“61440K”为老年代回收前的内存使用率，“41411K”为老年代回收后的内存使用率，“71680K”为老年代的总内存大小。“82849K->41411K(98560K)”代表着 Java 堆区的整体内存使用率。“[PSPermGen: 1616K->1614K(12288K)]”代表着永久代的内存使用率，“1616K”为永久代回收前的内存使用率，“1614K”为永久代回收后的内存使用率，“12288K”为永久代的总内存大小。“0.0467510 secs”代表着本次执行 Full GC 的耗时。

4．CMS 收集器的日志形式：

```
  [GC [1 CMS-initial-mark: 61892K(81920K)] 61892K(100352K), 0.0001474 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
  Total time for which application threads were stopped: 0.0397105 seconds
  [CMS-concurrent-mark: 0.005/0.005 secs] [Times: user=0.00 sys=0.00,
real=0.00 secs]
  [CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00,
real=0.00 secs]
  [GC[YG occupancy: 173 K (18432 K)][Rescan (parallel) , 0.0001639 secs][weak
refs processing, 0.0000073 secs][scrub string table, 0.0000315 secs]
[1 CMS-remark: 61892K(81920K)] 62066K(100352K), 0.0003017 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
  Total time for which application threads were stopped: 0.0004123 seconds
  [CMS-concurrent-sweep: 0.000/0.000 secs] [Times: user=0.00 sys=0.00,
real=0.00 secs]
  [CMS-concurrent-reset: 0.002/0.002 secs] [Times: user=0.00 sys=0.00,
real=0.00 secs]
```

很明显 CMS 收集器与之前另外几款垃圾收集器打印的 GC 日志在形式上大有不同。大家回想一下，笔者曾经在 6.6.4 节中对 CMS 收集器的执行过程进行过详细的讲解，那么为

了方便阅读，我们可以将 CMS 的 GC 日志划分为初始标记（Initial-Mark）、并发标记（Concurrent-Marking）、再次标记（Remark）、并发清除（Concurrent-Sweep）这 4 个主要阶段进行理解。

首先 CMS 会以一个称之为初始标记（Initial-Mark）阶段作为开始，在这个阶段中程序中所有的工作线程都将会因为“Stop-the-World”机制而出现约 0.0397105 秒的暂停（如果希望 GC 日志中能够打印出程序的暂停时间，则可以使用选项“-XX:+PrintGCApplication-StoppedTime”进行设置）。这个阶段的主要任务就是标记出内存中那些被根对象集合所连接的目标对象是否可达，最终执行耗时 0.0001474 秒。

接下来将会进入到并发标记（Concurrent-Marking）阶段，这个阶段的主要任务就是将之前的不可达对象标记为垃圾对象。由于该阶段程序中的工作线程会和垃圾收集线程同时运行或者交叉运行，所以该阶段并不会出现暂停，最终执行耗时 0.005 秒。

在正式执行并发清除（Concurrent-Sweep）之前，CMS 会对之前的垃圾对象再次标记(Remark)，以确保之前被标记为无用对象的引用关系不会遭到更改。和初始标记（Initial-Mark）阶段一样，该阶段同样也会因为“Stop-the-World”机制而出现约 0.0004123 秒的暂停，最终执行耗时 0.0003017 秒。

当经历过初始标记(Initial-Mark)、并发标记（Concurrent-Marking）和再次标记（Remark）三个阶段之后，CMS 最终将会进入到并发清除（Concurrent-Sweep）阶段执行内存回收，释放掉无用对象所占用的内存空间。“0.000/0.000”代表着并发清除耗时 CPU 的时间和墙上时间。如果排除并发预处理（Concurrent-Preclean）和并发重置（Concurrent-Reset）等阶段，当执行完这 4 个主要阶段后，CMS 收集器的回收任务也就算是完成了。

5．G1 收集器的日志形式：

```
[GC pause (young) (initial-mark), 0.00182844 secs]
   [Parallel Time:   1.4 ms]
      [GC Worker Start (ms):  159.5  159.5  159.5  159.6
       Avg: 159.5, Min: 159.5, Max: 159.6, Diff:   0.2]
      [Ext Root Scanning (ms):  0.6  0.2  0.5  0.3
       Avg:   0.4, Min:   0.2, Max:   0.6, Diff:   0.3]
      [Update RS (ms):  0.0  0.0  0.0  0.0
       Avg:   0.0, Min:   0.0, Max:   0.0, Diff:   0.0]
         [Processed Buffers : 0 0 0 6
          Sum: 6, Avg: 1, Min: 0, Max: 6, Diff: 6]
      [Scan RS (ms):  0.0  0.0  0.0  0.0
       Avg:   0.0, Min:   0.0, Max:   0.0, Diff:   0.0]
      [Object Copy (ms):  0.8  1.1  0.8  0.8
       Avg:   0.9, Min:   0.8, Max:   1.1, Diff:   0.3]
      [Termination (ms):  0.0  0.0  0.0  0.0
```

```
       Avg:   0.0, Min:   0.0, Max:   0.0, Diff:   0.0]
         [Termination Attempts : 2 1 3 2
          Sum: 8, Avg: 2, Min: 1, Max: 3, Diff: 2]
      [GC Worker End (ms):  160.8  160.8  160.8  160.8
       Avg: 160.8, Min: 160.8, Max: 160.8, Diff:   0.0]
      [GC Worker (ms):  1.4  1.3  1.3  1.2
       Avg:   1.3, Min:   1.2, Max:   1.4, Diff:   0.2]
      [GC Worker Other (ms):  0.1  0.1  0.1  0.3
       Avg:   0.1, Min:   0.1, Max:   0.3, Diff:   0.2]
   [Clear CT:   0.1 ms]
   [Other:   0.3 ms]
      [Choose CSet:   0.0 ms]
      [Ref Proc:   0.3 ms]
      [Ref Enq:   0.0 ms]
      [Free CSet:   0.0 ms]
   [Eden: 1024K(80M)->0B(79M) Survivors: 0B->1024K Heap: 80M(200M)->80M(200M)]
 [Times: user=0.06 sys=0.00, real=0.00 secs]
[GC concurrent-root-region-scan-start]
[GC concurrent-root-region-scan-end, 0.0005536]
[GC concurrent-mark-start]
[GC concurrent-mark-end, 0.0000599 sec]
[GC remark [GC ref-proc, 0.0000138 secs], 0.0011348 secs]
 [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC cleanup 180M->180M(200M), 0.0004300 secs]
 [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC pause (young), 0.00114628 secs]
   [Parallel Time:   0.9 ms]
      [GC Worker Start (ms):  228.1  228.1  228.1  228.3
       Avg: 228.1, Min: 228.1, Max: 228.3, Diff:   0.2]
      [Ext Root Scanning (ms):  0.3  0.3  0.2  0.2
       Avg:   0.3, Min:   0.2, Max:   0.3, Diff:   0.1]
      [Update RS (ms):  0.0  0.0  0.0  0.0
       Avg:   0.0, Min:   0.0, Max:   0.0, Diff:   0.0]
         [Processed Buffers : 2 0 0 0
          Sum: 2, Avg: 0, Min: 0, Max: 2, Diff: 2]
      [Scan RS (ms):  0.0  0.0  0.0  0.0
       Avg:   0.0, Min:   0.0, Max:   0.0, Diff:   0.0]
      [Object Copy (ms):  0.5  0.5  0.6  0.5
       Avg:   0.5, Min:   0.5, Max:   0.6, Diff:   0.1]
      [Termination (ms):  0.0  0.0  0.0  0.0
       Avg:   0.0, Min:   0.0, Max:   0.0, Diff:   0.0]
         [Termination Attempts : 1 1 1 1
          Sum: 4, Avg: 1, Min: 1, Max: 1, Diff: 0]
      [GC Worker End (ms):  228.9  228.9  228.9  228.9
       Avg: 228.9, Min: 228.9, Max: 228.9, Diff:   0.0]
      [GC Worker (ms):  0.8  0.8  0.8  0.7
       Avg:   0.8, Min:   0.7, Max:   0.8, Diff:   0.2]
      [GC Worker Other (ms):  0.1  0.1  0.1  0.3
       Avg:   0.2, Min:   0.1, Max:   0.3, Diff:   0.2]
```

```
    [Clear CT:   0.1 ms]
    [Other:   0.1 ms]
       [Choose CSet:   0.0 ms]
       [Ref Proc:   0.1 ms]
       [Ref Enq:   0.0 ms]
       [Free CSet:   0.0 ms]
    [Eden: 1024K(79M)->0B(79M) Survivors: 1024K->1024K
      Heap: 180M(200M)->180M(200M)]
  [Times: user=0.00 sys=0.00, real=0.00 secs]
  [Times: user=0.02 sys=0.00, real=0.01 secs]
```

为了节省篇幅，G1 收集器所打印的 GC 日志内容则可以参考 Oracle 的官方文档来进行理解，本书将不再过多讲解。

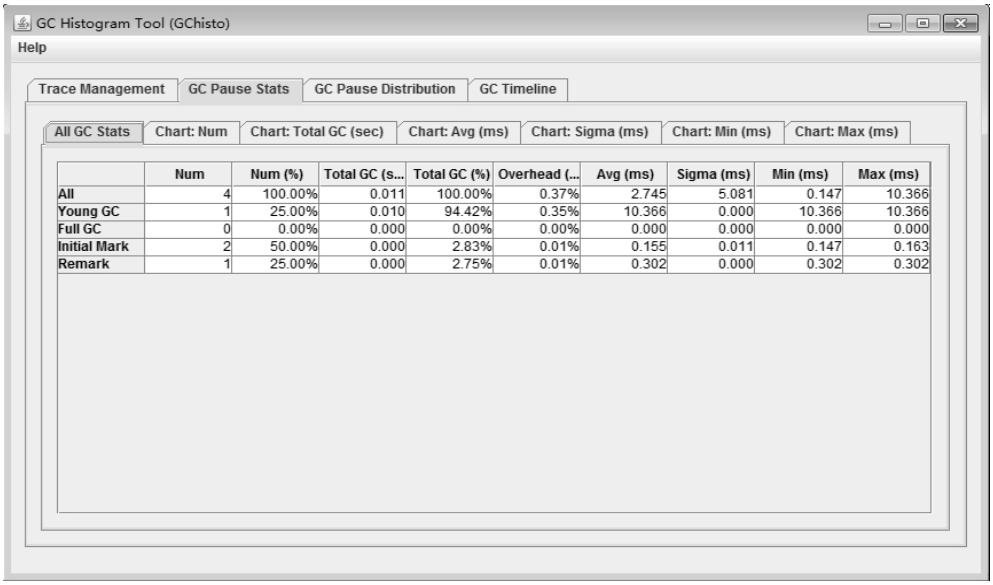
6.7.2 使用GCHisto工具分析离线日志

在生产环境中如果直接根据 GC 日志来排查故障或作为 JVM 性能调优的依据将会是一件极其困难的任务。通常情况下生产环境中的日志量非常大，仅凭肉眼来进行分析和判断并不直观，所以借助一些图形化界面工具来帮助我们分析 GC 日志将会是不错的选择。

除了可以对 GC 日志做离线分析外，还有一些专门用于实时分析的图形化界面工具。总之借助这些外围工具来帮助开发人员分析和定位问题不仅能够缩短时间，更重要的是，分析结果将会更加精确和直观。目前比较常用的几款 GC 日志分析工具有 GCHisto、GCLogViewer、HPjmeter 和 GCViewer 等。笔者最常用的还是 GCHisto 工具，所以接下来便以 GCHisto 工具为例进行讲解。GCHisto 是早期 Sun 公司提供的一款基于 Java 语言编写的开源离线 GC 日志分析工具，大家可以登录 https://java.net/projects/gchisto 下载 GCHisto 的源码。

当大家下载好 GCHisto 工具的源码并成功编译后，可直接运行/gchisto 目录下的 Main 类，或者将整个工程打包为一个可以运行的 jar 构件，当然究竟选择哪一种方式运行 GCHisto 则可以自行决定。GCHisto 启动并成功导入 gc.log 后，我们便可以根据统计界面生成的统计结果来进行分析。如图 6-14 所示。

在 GCHisto 工具的统计界面中，包含了总共执行 GC 的次数、Minor GC 的次数、Full GC 的次数、因“Stop-the-World”机制而导致程序暂停的初始标记和再次标记次数、GC 执行次数的百分比、执行 GC 的耗时、GC 的平均耗时和最小/最大耗时等。当然除了统计结果外，还可以查阅与这些统计结果相对应的图形化分析结果。如图 6-15 所示。



	Num	Num (%)	Total GC (s...	Total GC (%)	Overhead (...	Avg (ms)	Sigma (ms)	Min (ms)	Max (ms)
All	4	100.00%	0.011	100.00%	0.37%	2.745	5.081	0.147	10.366
Young GC	1	25.00%	0.010	94.42%	0.35%	10.366	0.000	10.366	10.366
Full GC	0	0.00%	0.000	0.00%	0.00%	0.000	0.000	0.000	0.000
Initial Mark	2	50.00%	0.000	2.83%	0.01%	0.155	0.011	0.147	0.163
Remark	1	25.00%	0.000	2.75%	0.01%	0.302	0.000	0.302	0.302

图 6-14 GCHisto 工具的统计界面

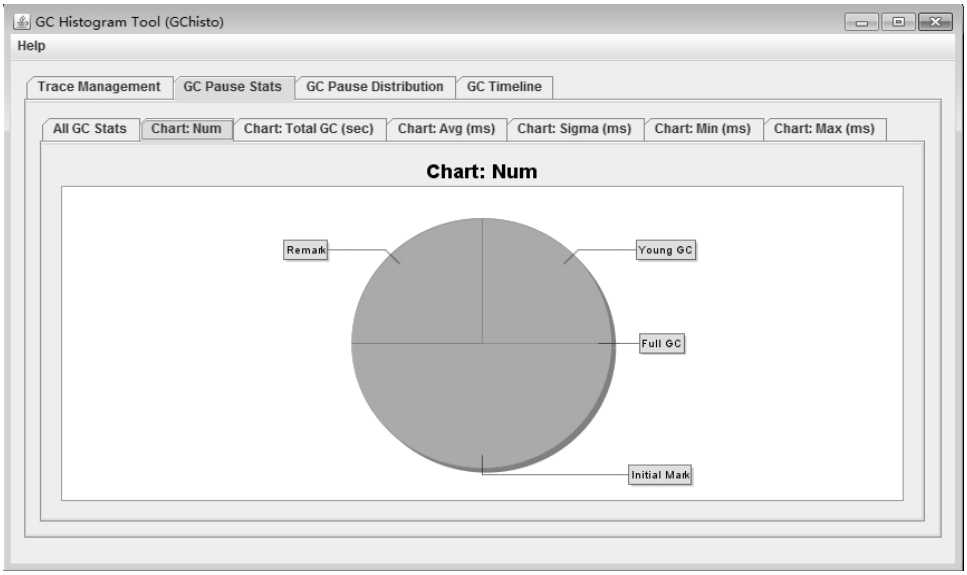


图 6-15 GCHisto 的图形化分析结果

使用 GCHisto 工具确实能够非常方便地根据 gc.log 迅速生成精准的统计结果和图形化分析结果，并且由于 GCHisto 工具是开放源代码的，所以如果大家感兴趣，完全可以随意在其功能基础之上进行功能扩展。

6.8 实战：分析dump文件

dump 文件实际上就是一份内存镜像文件，它记录了 Java 虚拟机在某一个时刻整个堆空间中的内存使用情况。在实际的开发过程中，开发人员可以根据 dump 文件来进行分析和排查程序在运行过程中是否发生 OutOfMemory 等情况。一般来说，生成 dump 文件最常见的手段就是通过使用 JDK 自带的 jmap 工具，再结合 JDK 自带的 jhat 工具、IBM HeapAnalyzer 工具或者 MAT（Memory Analyzer Tool）等工具来对其进行分析。

6.8.1 使用jmap工具生成dump文件

jmap 是 JDK 自带的一款专门用于生成 dump 文件的工具，通过使用该工具能够快速的生成 dump 文件。开发人员可以在控制台中输入命令“jmap -help”查阅 jmap 工具的具体使用方式和一些标准选项配置。如下所示：

具体用法:

```
jmap [option] <pid>
jmap [option] <executable <core>
jmap [option] [server_id@]<remote server IP or hostname>
```

其中选项包括:

-heap	输出整个堆空间的详细信息，包括 GC 的使用、堆配置信息，以及内存的使用信息等；
-histo	输出堆空间中对象的统计信息，包括类、实例数量和合计容量；
-permstat	以 ClassLoader 为统计口径输出永久代的内存状态信息；
-finalizerinfo	输出等待终结的对象信息；
-dump	生成 dump 文件；
-F	当虚拟机进程对-dump 选项没有任何响应时，强制执行生成 dump 文件；
-h | -help	jmap 工具的使用帮助命令；
-J<flag>	将 flag 直接传递给正在运行的虚拟机。

在此大家需要注意，jmap 工具有一部分选项仅限于在 Linux 和 Solaris 平台下使用，而 Windows 平台下能够使用的选项只有“-histo”和“-dump”。一般来说，用于生成 dump 文件的“jmap -dump:live,format=b,file=heap.bin <pid>”算得上是最常用的 jmap 命令之一。由于生成 dump 文件比较耗时，因此大家需要耐心等待，尤其是大内存镜像生成 dump 文件则需要耗费更长的时间来完成。

6.8.2 使用MAT（Memory Analyzer Tool）工具分析dump文件

之前笔者曾经提及过，JDK 自带的 jhat 工具就可以对 dump 文件进行分析。不过在实际的开发过程中，jhat 工具的使用率其实并不高，因为 jhat 工具的分析结果不直观，所以大部分开发人员往往选择使用 IBM HeapAnalyzer 工具或者 MAT（Memory Analyzer Tool）工具来对 dump 文件进行分析。那么本章的实战小节笔者就以 MAT 工具为例，为大家讲解如何使用它来对 dump 文件进行分析。

MAT 工具是 Eclipse 提供的，这就意味着 MAT 工具不仅可以单独使用，还可以作为插件的形式嵌入在 Eclipse 中使用，大家可以登录 http://www.eclipse.org/mat/downloads.php 下载 MAT 工具的最新版本。MAT 工具使用起来非常方便，尤其是在分析大内存的 dump 文件时，可以非常直观地看到各个对象在堆空间中所占用的内存大小、类实例数量、对象引用关系、利用 OQL 对象查询，以及可以很方便地找出对象 GC Roots 的相关信息，方便开发人员定位问题和分析问题。

当成功下载好 MAT 工具后，运行“\MemoryAnalyzer-1.4.0.20140604-win32.win32.x86\mat\”目录下的 MemoryAnalyzer.exe 即可成功启动 MAT。为了方便后续的演示，笔者提供了一段测试代码，如下所示：

代码 6-4 用于生成 dump 文件的测试代码

```
/**
 * -server -Xms1024m -Xmx1024m -Xmn384m -XX:+UseParallelOldGC
 *     -XX:+PrintGCApplicationStoppedTime     -XX:+PrintGCDateStamps
-XX:+PrintGCDetails
 * -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=D:/heap.bin
 *
 * @author JohnGao
 */
public class GCTest {
    public static void main(String[] args) {
        final int _1MB = 1024 * 1024;
        byte[] value1 = new byte[_1MB * 100];
        byte[] value2 = new byte[_1MB * 100];
        byte[] value3 = new byte[_1MB * 100];
        byte[] value4 = new byte[_1MB * 500];
        byte[] value5 = new byte[_1MB * 100];
    }
}
```

上述代码示例中，选项“-XX:+HeapDumpOnOutOfMemoryError”和-“XX:HeapDumpPath”所代表的含义就是当程序在运行时出现 OutofMemory 时，将会在指定的目录下生成一份当前 JVM 的堆内存 dump 文件，如果不指定选项“XX:HeapDumpPath”则会在当前目录下生

成 dump 文件。在此大家需要注意，尽管不借助 jmap 工具，MAT 工具也能够直接生成 dump 文件，但是考虑到生产环境中几乎不可能对其进行在线分析，大都采用离线分析，因此使用 jmap+MAT 工具是最常见的 dump 文件分析组合。

当执行代码 6-4 中的程序示例时，必然会发生 OutofMemory，然后在所指定的目录下找到生成的 dump 文件后，我们便可以通过 MAT 工具来进行分析了。当成功启动 MAT 后，通过菜单选项“File->Open heap dump”打开指定的 dump 文件将会生成 Overview 选项视图，如图 6-16 所示。

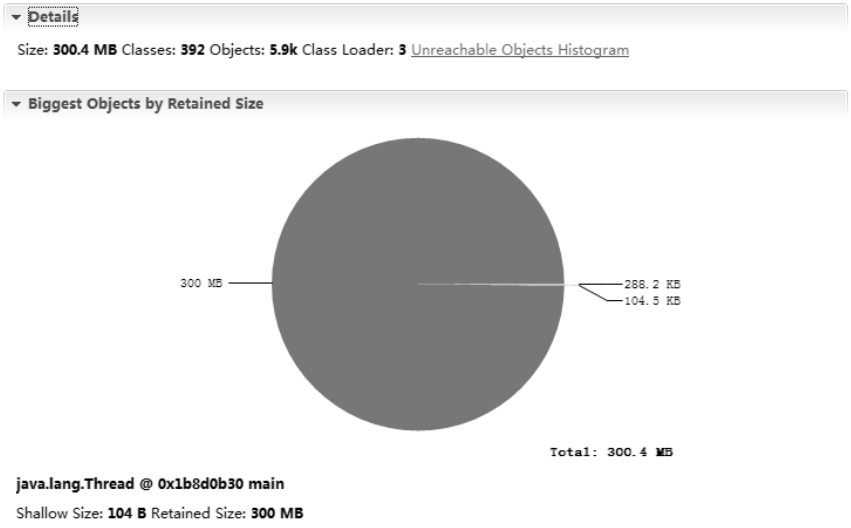
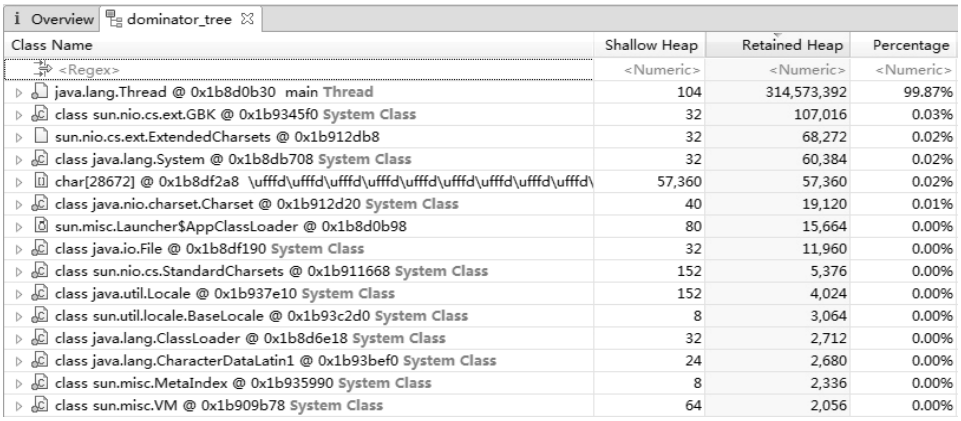


图 6-16　Overview 选项视图

在 Overview 选项视图中，以饼状图的形式列举了程序内存消耗的一些基本信息，其中每一种不同颜色的饼块都代表了不同比例的内存消耗情况。如果说需要定位出引发 OutofMemory 的代码点，我们则可以通过 Dominator Tree 菜单选项来进行排查（MAT 工具仅仅只是一种辅助手段，分析 OutofMemory 其实并不存在一个固定的方式和准则，因此仔细观察和分析才能找到问题所在），如图 6-17 所示。

在 Dominator Tree 选项视图中，内存中所有的对象都将会按照内存消耗的排名从高到低进行排序显示，由此可见代码 6-4 中引发 OutofMemory 的几个大对象都被标记在了首位。其中“Class Name”项是 Java 类的全限定名；“Shallow Heap”项和“Retained Heap”项分别指的是对象本身所消耗的内存大小，以及对象本身和它所引用的对象的内存大小总和；而“Percentage”项则用于表示对象消耗所占整个 dump total 的百分比。



Overview | dominator_tree

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.lang.Thread @ 0x1b8d0b30 main Thread	104	314,573,392	99.87%
class sun.nio.cs.ext.GBK @ 0x1b9345f0 System Class	32	107,016	0.03%
sun.nio.cs.ext.ExtendedCharsets @ 0x1b912db8	32	68,272	0.02%
class java.lang.System @ 0x1b8db708 System Class	32	60,384	0.02%
char[28672] @ 0x1b8df2a8 \ufffd\ufffd\ufffd\ufffd\ufffd\ufffd\ufffd\ufffd\ufffd\	57,360	57,360	0.02%
class java.nio.charset.Charset @ 0x1b912d20 System Class	40	19,120	0.01%
sun.misc.Launcher$AppClassLoader @ 0x1b8d0b98	80	15,664	0.00%
class java.io.File @ 0x1b8df190 System Class	32	11,960	0.00%
class sun.nio.cs.StandardCharsets @ 0x1b911668 System Class	152	5,376	0.00%
class java.util.Locale @ 0x1b937e10 System Class	152	4,024	0.00%
class sun.util.locale.BaseLocale @ 0x1b93c2d0 System Class	8	3,064	0.00%
class java.lang.ClassLoader @ 0x1b8d6e18 System Class	32	2,712	0.00%
class java.lang.CharacterDataLatin1 @ 0x1b93bef0 System Class	24	2,680	0.00%
class sun.misc.MetaIndex @ 0x1b935990 System Class	8	2,336	0.00%
class sun.misc.VM @ 0x1b909b78 System Class	64	2,056	0.00%

图 6-17 Dominator Tree 选项视图

6.9 本章小结

本章笔者首先详细地讲解了 Java 堆区、方法区、运行时常量池、PC 寄存器、Java 栈、本地方法栈以及性能监控区等 7 个在 HotSpot 中非常重要的运行时内存区后，紧接着对 HotSpot 的自动内存管理机制进行了深入的讲解，其中涵盖了对象的内存分配原理、逃逸分析和栈上分配、内存布局与 OOP-Klass 模型、垃圾对象如何标记，以及不同的垃圾收集算法。当大家理解自动内存管理机制的原理后，本章又介绍了 Serial/Serial Old 收集器、ParNew 收集器、Parallel/Parallel Old 收集器、CMS 收集器和 G1 收集器等各自的特点。而在本章的实战小节中，笔者还讲解了如何对 GC 日志进行分析，以及如何使用 jmap+MAT 工具来分析 dump 文件。

第 7 章

类加载机制

任何一个类型在使用之前都必须经历过完整的加载、连接和初始化 3 个类加载步骤。一旦一个类型成功经历过这 3 个步骤之后，它就可以随时随地被使用了，开发人员可以在程序中访问和调用它的静态类成员信息（比如：静态字段、静态方法），或者使用 new 关键字为其创建对象实例。当然从一个类型被加载进 JVM 中开始算起，直至最终被卸载出内存为止，它的整个生命周期也就随之而结束了。

7.1 类加载器

在正式开始讲解类加载机制之前，笔者有必要首先为大家介绍一下类加载器，因为类加载器是 JVM 执行类加载机制的前提。简单来说，类加载器的主要任务就是根据一个类的全限定名来读取此类的二进制字节流到 JVM 内部，然后转换为一个与目标类对应的 java.lang.Class 对象实例。类加载器最早出现在 Java1.0 版本中，那个时候只是单纯地为了满足 Java Applet 应用而被研发出来，但如今类加载器却在 OSGi、字节码加解密领域大放异彩。这主要归功于 Java 虚拟机的设计者们当初在设计类加载器的时候，并没有考虑将它绑定在 JVM 内部，这样做的好处就是能够更加灵活和动态地执行类加载操作。

一般情况下，Java 开发人员并不需要在程序中显式地使用类加载器，但是了解类加载器的加载机制却显得至关重要。相信大部分 Java 开发人员几乎都在日常的编码工作中遇见过 java.lang.ClassNotFoundException 异常或 java.lang.NoClassDefFoundError 异常，所以为了避免在这类异常出现时发生手足无措的情况，了解类加载器的加载机制也就成为了每一个 Java 开发人员必须掌握的一门技能，因为只有这样才能够在出现异常的时候快速地根据错误异常日志定位问题和解决问题。除此之外，在一些特殊的应用场景中，比如需要支持类的动态加载或需要对编译后的字节码文件进行加解密操作时，就需要与类加载器打交道了，也就是说，开发人员可以在程序中编写自定义类加载器来重新定义类的加载规则，以便实现一

些自定义的处理逻辑。

参考《Java 虚拟机规范（Java SE7 版）》的描述来看，JVM 支持两种类型的类加载器，分别为引导类加载器（Bootstrap ClassLoader）和自定义类加载器（User-Defined ClassLoader）。在此大家需要注意，从概念上来讲，自定义类加载器一般指的是程序中由开发人员自定义的一类类加载器，但是 Java 虚拟机规范却没有这么定义，而是将所有派生于抽象类 ClassLoader 的类加载器都划分为自定义类加载器。无论类加载器的类型如何划分，在程序中我们最常见的类加载器始终只有 3 个，如下所示：

- Bootstrap ClassLoader；
- ExtClassLoader；
- AppClassLoader。

Bootstrap ClassLoader 也称之为启动类加载器，它由 C++语言编写并嵌套在 JVM 内部，主要负责加载“JAVA_HOME/lib”目录中的所有类型，或者由选项“-Xbootclasspath”指定路径中的所有类型。ExtClassLoader 和 AppClassLoader 派生于 ClassLoader，并且都是采用 Java 语言进行编写的，前者主要负责加载“JAVA_HOME/lib/ext”扩展目录中的所有类型，而后者则主要负责加载 ClassPath 目录中的所有类型。如下所示：

代码 7-1 不同类加载器的加载类型

```
/**
 * 不同类加载器的加载类型
 *
 * @author JohnGao
 */
public class ClassLoaderTest {
    public static void main(String[] args) {
        /* Bootstrap ClassLoader 负责加载 JAVA_HOME\lib 目录中的类型 */
        ClassLoader loader = System.class.getClassLoader();
        System.out.println(null != loader ? loader.getClass().
          getName() : null);

        /* ExtClassLoader 负责加载 JAVA_HOME\lib\ext 目录中的类型 */
        System.out.println(CollationData_ar.class.getClassLoader().
          getClass().getName());

        /* AppClassLoader 负责加载 ClassPath 目录中的类型 */
        System.out.println(ClassLoaderTest.class.getClassLoader().
          getClass().getName());
    }
}
```

程序输出如下：

```
null
sun.misc.Launcher$ExtClassLoader
sun.misc.Launcher$AppClassLoader
```

上述代码示例中，笔者通过 getClassLoader()方法成功获取了目标类的类加载器，返回值就是一个 ClassLoader 的派生实例，但这个方法却并非属于 ClassLoader，而属于 java.lang.Class。java.lang.System 类包含在“JAVA_HOME/lib”目录中，由 Bootstrap ClassLoader 负责加载，但是程序最终输出的类加载器名称却是 Null，当然这并不代表类加载器不存在，而是因为启动类加载器本身是由 C++语言编写并嵌套在 JVM 内部的，所以输出结果才会为 Null。sun.text.resources.CollationData_ar 类包含在“JAVA_HOME/lib/ext”扩展目录中，由 ExtClassLoader 负责加载。而 ClassLoaderTest 类是笔者自己编写的一个 Java 类，包含在 ClassPath 目录中，因此由 AppClassLoader 负责加载。

7.1.1 抽象类ClassLoader

如果当前的类加载器无法满足我们的需求时，便可以在程序中编写自定义类加载器来重新定义类的加载规则，以便实现一些自定义的处理逻辑。在程序中编写一个自定义类加载器是一件非常简单的任务，只需要继承抽象类 ClassLoader 并重写其 findClass()方法即可。当编写好自定义类加载器后，便可以在程序中调用 loadClass()方法来实现类加载操作。

在编写一个自定义类加载器的时候，我们经常会用到 ClassLoader 中的一些比较常用的方法，那么接下来笔者将会围绕这些方法进行讲解。如表 7-1 所示。

表 7-1 ClassLoader 的常用方法

方法名称	描述
getParent()	返回该类加载器的超类加载器
loadClass(String name)	加载名称为 name 的类，返回结果为 java.lang.Class 类的实例
findClass(String name)	查找名称为 name 的类，返回结果为 java.lang.Class 类的实例
findLoadedClass(String name)	查找名称为 name 的已经被加载过的类，返回结果为 java.lang.Class 类的实例
defineClass(String name,byte[] b,int off,int len)	把字节数组 b 中的内容转换为一个 Java 类，返回结果为 java.lang.Class 类的实例
resolveClass(Class<?> c)	连接指定的一个 Java 类

在表 7-1 中，ClassLoader 的 getParent()方法用于获取当前类加载器的超类加载器。按照双亲委派模型（请阅读 7.1.2 节）的规则，除了启动类加载器之外，程序中每一个类加载器都应该拥有一个超类加载器，比如 AppClassLoader 的超类加载器就是 ExtClassLoader。如下所示：

代码 7-2 使用 getParent()方法获取当前类加载器的超类加载器

```
/**
 * 获取当前类加载器的超类加载器
 *
 * @author JohnGao
 */
public class GetSuperClassLoader {
    public static void main(String[] args) {
            /* 获取目标类的类加载器 */
        ClassLoader classLoader = GetSuperClassLoader.class.
getClassLoader();
        System.out.println("当前类加载器->" + classLoader.getClass().
getName());
        /* 获取超类加载器 */
        classLoader = classLoader.getParent();
        System.out.println("当前类加载器的超类加载器->" + classLoader.
getClass().getName());
    }
}
```

程序输出如下：

```
当前类加载器->sun.misc.Launcher$AppClassLoader
当前类加载器的超类加载器->sun.misc.Launcher$ExtClassLoader
```

在程序中，一个普通的 Java 类一旦继承抽象类 ClassLoader，并重写其 findClass()方法后，它就已经是一个自定义类加载器了。一般情况下，findClass()和 defineClass()通常是一起组合使用的。在重写 findClass()方法时，可以指定相应的逻辑处理操作，比如，如果一个字节码文件在编译的时候进行了加密处理，以防止其他人进行反编译，那么这个特殊的字节码文件在被加载进 JVM 内部之前，首先需要做的就是执行解密操作，当成功解密之后，就可以调用 defineClass()方法将解密后的 byte 数组转换为一个类的 Class 对象实例。

使用 defineClass()读取字节码的数据源不仅限于磁盘，还可以是网络，总之只要是能够成功获取一个类的字节码，就都能通过 defineClass()方法将其转换为一个类的 Class 对象实例。在此大家需要注意，如果希望在类被加载到 JVM 内部时就被链接（Link），那么便可以调用 resolveClass()方法，当然也可以由 JVM 来选择应该在什么时候才执行链接操作。

7.1.2 双亲委派模型

笔者在 7.1 节中为大家介绍了究竟什么是类加载器，在程序中最常见的 3 种类加载器，分别为 Bootstrap ClassLoader、ExtClassLoader 和 AppClassLoader。当然开发人员也可以在程序中编写一些自定义类加载器来与这些默认的类加载器共同协作执行类加载任务。大家思

考一下，在执行类加载任务的时候，如何确保一个类的全局唯一性？举个贴近生活一点的例子。小娇前些天在一家手机专营店购买了一台某个品牌的智能手机，但是在使用的过程中由于非人为损坏导致手机无法正常使用，于是小娇准备拿着手机保修单，要求商家按照保修单上的承诺免费为其更换一台全新的同款手机。当小娇来到所购手机的专营店后，商家只是审核了手机之前的退换货记录（手机只能免费更换一次），但没有立即给予小娇明确答复，也没有对手机进行损坏检测，而是直接将小娇的情况反映给了这家手机专营店的总店，并告知小娇前往总店商量相关的退换货事宜。尽管非常麻烦和极度不情愿，小娇还是按照商家的要求来到了总店，当总店的维修人员对手机做了详细的损坏检测后，最终同意为小娇免费更换一台全新的同款智能手机。但由于小娇并非是在总店处购买的，因此只能前往分店进行更换，于是小娇辗转反侧又重新回到了手机的购买处，当填写完相关的退换货资料并存档后，小娇终于高兴地拿着新手机回家了。

尽管上述故事非常无聊，但是这却和类加载器的加载机制非常相似。Java 虚拟机的设计者们通过一种被称之为双亲委派模型（Parents Delegation Model）的委派机制来约定类加载器的加载机制。按照双亲委派模型的规则，除了启动类加载器之外，程序中每一个类加载器都应该拥有一个超类加载器，比如 AppClassLoader 的超类加载器就是 ExtClassLoader，而开发人员自己编写的自定义类加载器的超类就是 AppClassLoader。那么当一个类加载器接收到一个类加载任务的时候，它并不会立即展开加载，而是将加载任务委派给它的超类加载器去执行，每一层的类加载器都采用相同的方式，直至委派给最顶层的启动类加载器为止。如果超类加载器无法加载委派给它的类时，便会将类的加载任务退回给它的下一级类加载器去执行加载。如图 7-1 所示。

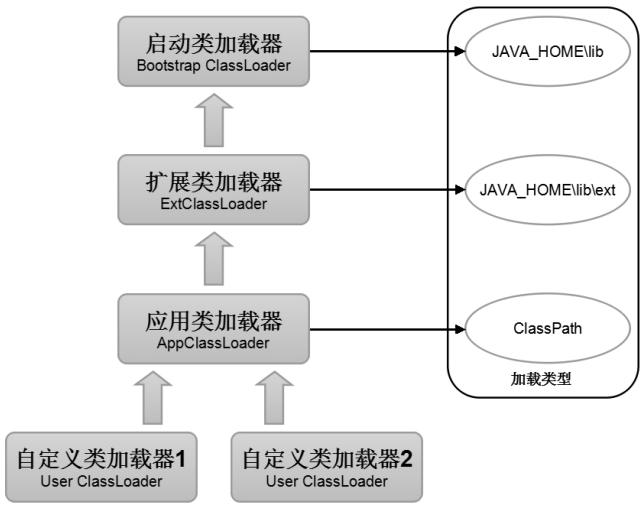


图 7-1　双亲委派模型

使用双亲委派模型的优点就是能够有效地确保一个类的全局唯一性，当程序中出现多个全限定名相同的类时（比如都叫做 com.test.User），类加载器在执行加载的时候，始终都只会加载其中的某一个类，不会 2 个类都执行加载，如果想通过 defindClass()方法进行显示的加载，JVM 将会抛出异常。为了方便阅读，本书示例了包含在抽象类 ClassLoader 中的 loadClass()方法，该方法中的内容就是双亲委派模型在语法层面上的具体实现。如下所示：

代码 7-3 双亲委派模型的具体实现

```
    protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException
    {
        synchronized (getClassLoadingLock(name)) {
            /* 首先检查目标类型之前是否已经被成功加载过了 */
            Class c = findLoadedClass(name);
            if (c == null) {
                long t0 = System.nanoTime();
                try {
                    /* 如果存在超类加载器，就委派给超类加载器执行加载 */
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        /* 如果不存在超类加载器，就直接委派给顶层的启动类加载器执
行加载 */
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    /* 当抛出 ClassNotFoundException 异常时，则意味着超类加载器
加载失败 */
                }

                if (c == null) {
                    /* 如果超类加载器无法加载时，则自行加载 */
                    long t1 = System.nanoTime();
                    c = findClass(name);

                    // this is the defining class loader; record the stats
        sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
        sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                    sun.misc.PerfCounter.getFindClasses().increment();
                }
            }
            if (resolve) {
                resolveClass(c);
            }
            return c;
        }
    }
```

上述代码示例中，首先会由 findLoadedClass()方法检查目标类型之前是否已经被成功加载过了，如果确定没有被加载过，则调用超类加载器的 loadClass()方法，将类的加载任务委派给它的超类加载器，如果不存在超类加载器，就通过 findBootstrapClassOrNull()方法调用本地方法 findBootstrapClass()直接委派给顶层的启动类加载器去执行加载。如果超类加载器抛出 ClassNotFoundException 异常时，则意味着超类加载器无法加载目标类型，那么则只能够调用自身的 findClass()方法执行类加载操作。

在此大家需要注意，由于 Java 虚拟机规范并没有明确要求类加载器的加载机制一定要使用双亲委派模型，只是建议采用这种方式而已。比如在 Tomcat 中，类加载器所采用的加载机制就和传统的双亲委派模型有一定区别，当缺省的类加载器接收到一个类的加载任务时，首先会由它自行加载，当它加载失败时，才会将类的加载任务委派给它的超类加载器去执行，这同时也是 Servlet 规范推荐的一种做法。

7.1.3 自定义类加载器

一般来说，在程序中开发人员使用自己编写的自定义类加载器来与默认的 3 种类加载器共同执行类加载任务的情况其实并不多见，但是在一些特殊的应用场景中，如果当前的类加载器无法满足我们的需求时，就需要在程序中编写自定义类加载器来重新定义类的加载规则，以便实现一些自定义的处理逻辑。比如就像笔者之前提及过的，当一个字节码文件在编译的时候如果进行了加密处理，那么在被类加载器执行类加载操作时，首先要做的事情就是解密处理，否则类加载器就会认为这个字节码并不符合 JVM 规范，它不是一个有效的字节码文件。其次就是程序中如果没有显式地指定类加载器时，AppClassLoader 就是任务委派的发起者。AppClassLoader 主要负责加载 ClassPath 目录中的所有类型，但是如果被加载的类型并没有包含在 ClassPath 目录中时，程序最终就会抛出 java.lang.ClassNotFoundException 异常。为了满足这些特殊的应用场景，开发人员就需要在程序中编写自定义类加载器。

想要在程序中实现一个自定义类加载器是非常简单的，只需要继承抽象类 ClassLoader，并重写其 findClass()方法即可。在此大家需要注意，尽管 Java 虚拟机规范将所有派生于抽象类 ClassLoader 的类加载器都划分为自定义类加载器，但是从严格意义上来说，由 Java 开发人员编写的自定义类加载其实并不属于 Java 体系结构的组成部分，实际上它仅仅只是属于 Java 运行时程序的一部分而已。如下所示：

代码 7-4 编写自定义类加载器

```
/**
 * 自定义类加载器
 *
 * @author JohnGao
 */
```

```
public class MyClassLoader extends ClassLoader {
    private String byteCode_Path;

    public MyClassLoader(String byteCode_Path) {
        this.byteCode_Path = byteCode_Path;
    }

    @Override
    protected Class<?> findClass(String className)
            throws ClassNotFoundException {
        byte value[] = null;
        BufferedInputStream in = null;
        try {
            in = new BufferedInputStream(new FileInputStream
                    (byteCode_Path + className + ".class"));
            value = new byte[in.available()];
            in.read(value);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            /* 释放资源 */
            if (null != in) {
                try {
                    in.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        /* 将 byte 数组转换为一个类的 Class 对象实例 */
        return defineClass(value, 0, value.length);
    }

    public static void main(String[] args) throws Exception {
        MyClassLoader classLoader = new MyClassLoader("D:/");
        System.out.println("加载目标类的类加载器->"
                + classLoader.loadClass("Test").getClassLoader()
                        .getClass().getName());
        System.out.println("当前类加载器的超类加载器->"
                + classLoader.getParent().getClass().getName());
    }
}
```

程序输出如下：

```
加载目标类的类加载器->MyClassLoader
当前类加载器的超类加载器->sun.misc.Launcher$AppClassLoader
```

上述代码示例中，笔者演示了如何编写一个自定义类加载器来加载指定目录中的类型，

由于目标类型 Test 并非包含在 ClassPath 目录中，自然也就无法被 AppClassLoader 执行加载。在程序中指定了加载目标类型的类加载器是 MyClassLoader，那么它就是任务委派的发起者，MyClassLoader 会按照双亲委派模型将 Test 类的加载任务委派给超类加载器，而 AppClassLoader 将会按照这种方式最终委派给顶层的启动类加载器，只有当超类加载器无法执行加载时，才会将类加载任务退回给 MyClassLoader 执行。笔者在重写 findClass()方法时指定了以流的形式读取目标类型的二进制字节流到 JVM 内部，然后通过 defineClass()方法将其转换为一个类的 Class 对象实例。

7.1.4 定位ClassNotFoundException异常

对于 Java 开发人员而言，java.lang.ClassNotFoundException 恐怕是在实际开发过程中出现频率最高的异常之一。一般情况下，这类异常往往是因为开发人员在程序中显式地使用了 Class.forName()方法或者 ClassLoader.getSystemClassLoader().loadClass()方法显式地加载一个类型所引起的，那么如果程序中一旦出现了该异常时，也就意味着类加载器执行类加载任务失败，因为类加载器无法根据所指定的全限定名正确地加载到目标类型。

在代码 7-5 示例中，笔者使用了 Class.forName()方法来显式加载一个全限定名不存在的类型时，程序在运行时将会抛出 java.lang.ClassNotFoundException 异常。如下所示：

代码 7-5 触发 ClassNotFoundException 异常

```
/**
 * 触发 ClassNotFoundException 异常
 *
 * @author JohnGao
 */
public class ClassNotFoundTest {
    public static void main(String[] agrs) {
        try {
            Class.forName("java.lang.Object_");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

程序输出如下：

```
  Exception in thread "main" java.lang.ClassNotFoundException: java.lang.
Object_
    at java.net.URLClassLoader$1.run(URLClassLoader.java:366)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
    at java.security.AccessController.doPrivileged(Native Method)
```

```
    at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:423)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:356)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:188)
    at ClassNotFoundTest.main(ClassNotFoundTest.java:8)
```

上述代码示例中，笔者在 forName()方法中指定了一个全限定名为 java.lang.Object_的类型，只不过该类型并不存在，那么当 AppClassLoader 在 ClassPath 目录中找不到该类型时（开发人员可以在程序中使用 System.getProperty(“java.class.path")方法查阅当前的 ClassPath 目录），就意味着加载任务执行失败，最终程序就会抛出 java.lang.ClassNotFoundException 异常。

7.1.5 定位NoClassDefFoundError异常

和 java.lang.ClassNotFoundException 异常一样，java.lang.NoClassDefFoundError 同样也是在实际开发过程中出现频率最高的异常之一。

在代码 7-6 示例中，笔者引用了 Spring 的一些相关构件用以初始化 IoC 容器，尽管能够成功通过编译，但是程序在运行的过程中，如果无法正确引用到所依赖的相关 Spring 构件时，将会抛出 java.lang.ClassNotFoundException 异常。如下所示：

代码 7-6 触发 NoClassDefFoundError 异常

```
/**
 * 触发 NoClassDefFoundError 异常
 *
 * @author JohnGao
 */
public class NoClassDefFoundTest {
    public static void main(String[] args) {
        /* 加载 Spring 的 Ioc 配置文件，并初始化 Ioc 容器 */
        new org.springframework.context.support.
            ClassPathXmlApplicationContext("root-context.xml");
    }
}
```

程序输出如下：

```
  Exception in thread "main" java.lang.NoClassDefFoundError:
 org/springframework/context/support/ClassPathXmlApplicationContextat
 NoClassDefFoundTest.main(NoClassDefFoundTest.java:10)Caused by:java.lang.
ClassNotFoundException:
 org.springframework.context.support.ClassPathXmlApplicationContext
```

```
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
    at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
... 1 more
```

当上述代码成功通过编译后，在使用命令“java NoClassDefFoundTest”执行之前，只要“切断”程序与第三方构件之间的依赖时，就会抛出 java.lang.NoClassDefFoundError 异常。所以大家在实际的开发过程中，尤其是在生产环境中执行项目部署时，一定要非常细心，确保项目中所依赖的所有第三方构件都能够被成功引用到。

7.2 类的加载过程

在前面几个小节中，大家已经知道了类加载器的主要任务就是根据一个类的全限定名来读取此类的二进制字节流到 JVM 内部，然后转换为一个与目标类对应的 java.lang.Class 对象实例。当然类加载器所执行的加载操作仅仅只是属于 JVM 中类加载过程中的一个阶段而已，一个完整的类加载过程必须经历加载、连接和初始化这 3 个步骤。如图 7-2 所示。

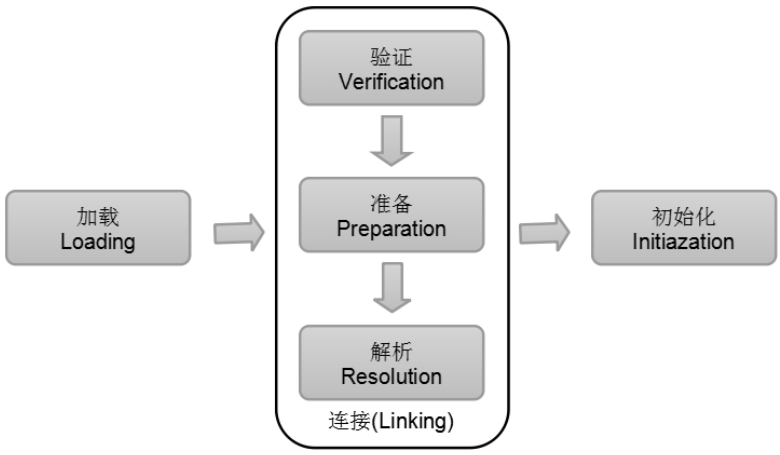


图 7-2 完整的类加载过程

简单来说，类的加载阶段就是由类加载器负责根据一个类的全限定名来读取此类的二进制字节流到 JVM 内部，并存储在运行时内存区中的方法区内，然后将其转换成一个与目标类型对应的 java.lang.Class 对象实例（Java 虚拟机规范并没有明确要求一定要存储在 Java 堆区中，因此 HotSpot VM 选择将 Class 对象存储在方法区内），这个 Class 对象在日后就会作为方法区中该类的各种数据的访问入口。而链接阶段要做的事情就是将已经加载到 JVM

中的二进制字节流的类数据信息合并到 JVM 的运行时状态中，然而连接阶段则由验证、准备和解析 3 个阶段构成，其中验证阶段的主要任务就是验证类数据信息是否符合 JVM 规范，是否是一个有效的字节码文件，而验证的内容则涵盖了类数据信息的格式验证、语义分析、操作验证等；准备阶段的主要任务就是为类中的所有静态变量分配内存空间，并为其设置一个初始值（由于还没有产生对象，因此实例变量将不再此操作范围内）；而解析阶段的主要任务就是将常量池中所有的符号引用全部转换为直接引用，不过 Java 虚拟机规范并没有明确要求解析阶段一定要按照顺序执行，因此解析阶段可以等到初始化之后再执行。类加载过程中的最后一个阶段就是初始化，在这个阶段中，JVM 会将一个类中所有被 static 关键字标示的代码统统执行一遍，如果执行的是静态变量，那么就会使用用户指定的值覆盖掉之前在准备阶段中 JVM 为其设置的初始值，当然如果程序中没有为静态变量显式指定赋值操作，那么所持有的值仍然是之前的初始值；反之如果执行的是 static 代码块，那么在初始化阶段中，JVM 就将会执行 static 代码块中定义的所有操作。

Java 虚拟机规范在类加载和连接的时机上提供了较大的灵活性，但 Java 虚拟机规范却明确规定了类的初始化时机，也就是说，一个类或者接口应该在首次主动使用时执行初始化操作，如下所示：

- 为一个类型创建一个新的对象实例时（比如使用 new 关键字、反射或序列化）；
- 调用一个类型的静态方法时（即在字节码中执行 invokestatic 指令）；
- 调用一个类型或接口的静态字段，或者对这些静态字段执行赋值操作时（即在字节码中，执行 getstatic 指令或 putstatic 指令），不过用 final 关键字修饰的静态字段除外，它被初始化为一个编译时的常量表达式；
- 调用 Java API 中的反射方法时（比如调用 java.lang.Class 中的方法，或者 java.lang.reflect 包中其他类的方法）；
- 初始化一个类的派生类时（Java 虚拟机规范明确要求初始化一个类时，它的超类必须提前完成初始化操作）；
- JVM 启动包含 main()方法的启动类时。

上述笔者所列出的 6 种情况都属于主动使用的情形，而其他的情形则都不属于主动使用，因此它们都不会导致一个类型被执行初始化操作。在此大家需要注意，尽管一个类在初始化之前必须要求它的超类提前完成初始化操作，但对于接口而言，这条规则却显得并不适用。只有在某个接口中声明的非常量字段被使用时，该接口才会被初始化，而不会因为实现这个接口的派生接口或派生类要初始化而被初始化，也就是说，接口并不要求在执行初始化操作的时候，它的超类接口必须提前完成初始化操作。

为了让大家更好地理解类加载机制，笔者将通过代码 7-7 示例来为大家演示一个类在执行类加载过程中所发生的一些微妙变化。如下所示：

代码 7-7　根据一个简单的程序示例剖析类加载机制

```
/**
 * 根据一个简单的程序示例剖析类加载机制
 *
 * @author JohnGao
 */
public class LoadingTest {
    public static LoadingTest obj = new LoadingTest();
    public static int value1;
    public static int value2 = 0;

    public LoadingTest() {
        value1 = 10;
        value2 = value1;
        System.out.println("before value1->" + value1);
        System.out.println("before value2->" + value2);
    }

    public static void main(String[] args) throws Exception {
        System.out.println("after value1->" + value1);
        System.out.println("after value2->" + value2);
    }
}
```

程序输出如下：

```
before value1->10
before value2->10
after value1->10
after value2->0
```

上述代码示例中，如果大家不细心的话，肯定会认为在 main()方法中打印的 Loading 类的静态变量 value1 和 value2 的值都为 1，但实际上程序最终输出的结果却是 value1 等于 10，而 value2 等于 0。在 Loading 类的构造方法中，尽管已经将静态变量 value1 和 value2 都显式赋值为 10 了，并且在构造方法中所打印的结果也的确为 10，那为什么会与 main()方法中输出的结果不一致呢？当笔者将上述代码的位置稍作调整后，程序最终的输出结果是否又会产生变化呢？如下所示：

代码 7-8　将 7-7 示例中的代码位置调换顺序

```
public static int value1;
public static int value2 = 0;
public static LoadingTest obj = new LoadingTest();
```

程序输出如下：

```
before value1->10
before value2->10
after value1->10
after value2->10
```

当笔者将声明静态变量 obj 的代码位置放在声明静态变量 value2 之后，程序最终的输出结果就跟预期的值一致了，这又是为什么呢？简单来说，当类加载器将 Loading 类加载进 JVM 内部后，会在方法区中生成一个与该类型对应的 java.lang.Class 对象实例，当进入到准备阶段时，JVM 便会为 Loading 类中的 3 个静态变量分配内存空间，并为其设置初始值（value1 和 value2 的初始值为 0，而 obj 的初始值则为 null）。当经历到类加载过程中的初始化阶段时，程序最终的输出结果就会和代码的执行顺序有关了。在代码 7-7 示例中，静态变量 obj 是优先初始化的，那么 JVM 将会执行到其构造方法中，并覆盖掉静态变量 value1 和 value2 之前持有的初始值，也就是说，初始化静态变量 obj 后，value1 和 value2 所持有的值就都是 10。接下来 JVM 会检查静态变量 value1 是否也需要执行初始化，由于 value1 并没有显式地指定进行赋值操作，因此将会直接跳转到静态变量 value2 上，这里就非常关键了，并且也是代码 7-7 示例中程序最终的输出结果与预期不一致的“罪魁祸首”。尽管之前指定了 value2 所持有的值为 10，但当执行到 value2=0 时等于又重新对静态变量 value2 显式执行了一次赋值操作，也就是说，当前的赋值操作将会覆盖掉之前在构造方法中的赋值操作，这就是为什么 value2 的输出结果为 0，而非为 10 的真正原因。然而在代码 7-8 示例中，笔者将代码的执行顺序进行了调换，尽管 value2 同样也经历过 2 次赋值操作，但最后一次被赋予的值却是 10，所以程序最终的输出结果就跟预期一致。

7.2.1 加载字节码

加载字节码是由类加载器所负责的，当然加载操作仅仅只是类加载过程中的一个阶段，同样也是类加载过程中的第一个阶段，而后续还需要连接和初始化阶段的配合才能构成一个完整的类加载过程。之前笔者也说过，类的加载阶段就是由类加载器负责根据一个类的全限定名来读取此类的二进制字节流到 JVM 内部，并存储在运行时内存区中的方法区内，然后将其转换成一个与目标类型对应的 java.lang.Class 对象实例（Java 虚拟机规范并没有明确要求一定要存储在 Java 堆区中，因此 HotSpot VM 选择将 Class 对象存储在方法区内），这个 Class 对象在日后就会作为方法区中该类的各种数据的访问入口。在此需要提醒大家，OPP-Klass 模型代表一个对象的目标类型则是由 Klass 中的 instanceKlass 对象表示（数组则用 arrayKlass 对象表示），不过 Klass 和语法层面的 java.lang.Class 并不能等价，因为 Klass 用于运行，而 Class 只用于 Java 语法层面的反射 API，Klass 中有_java_mirror 字段指向 Class。

如果大家继承 ClassLoader 并重写其 findClass()方法实现一个自定义类加载器后，应该

知道类加载器读取目标类的二进制字节流是没有限定的，这就意味着，开发人员不仅可以从一个字节码文件中读取，还能够从网络中读取，甚至可以从数据库中获取出一个类的二进制字节流等。JVM 虚拟机并不关心数据从哪里读取，只要求所读取的字节码符合 JVM 规范，因为在后续的验证阶段中，JVM 会验证所加载的二进制字节流包含的信息是否会对虚拟机本身产生影响。

参考《Java 虚拟机规范 Java SE7 版》的描述来看，创建数组类的情况稍微有些特殊，简单来说，数组类本身并不是由类加载器负责创建，而是由 JVM 在运行时根据需要而直接创建的，但数组的元素类型仍然需要依靠类加载器去创建。创建数组类（下述简称 C）的过程，如下所示：

- 如果数组的元素类型是引用类型，那么就遵循本节中定义的加载过程递归加载和创建数组 C 的元素类型；
- JVM 使用指定的元素类型和数组维度来创建新的数组类。

如果元素类型是引用类型，数组 C 就被标记为它已经被该元素类型的定义类加载器定义过。否则数组 C 就被标记为它被引导类加载器定义过。不管哪种情况，JVM 都会把类加载器记录为数组 C 的初始加载器。如果数组的元素类型是引用类型，数组类的可访问性就由元素类型的可访问决定，否则数组类的可访问性将被缺省定义为 public。

7.2.2 验证阶段

当一个类型成功被类加载器加载进 JVM 内部之后，接下来要做的事情就是执行链接操作，将已经成功加载到 JVM 中的类的二进制信息合并到 JVM 的运行时状态中去。验证操作属于链接中的第一个阶段，如图 7-2 所示，只有当一个类型被成功加载之后，才能够执行验证操作，但是 JVM 的设计者们并不一定会按照这种方式执行类加载。也就是说，类加载过程与 JVM 的具体实现息息相关，在 HotSpot VM 中，加载、验证、准备和初始化会按照顺序有条不紊地执行，但链接阶段中的解析操作往往会伴随着 JVM 在执行完初始化之后再执行。除此之外，验证阶段中的一些验证操作将会和加载阶段一起执行，而不会等待加载完成之后才会执行某些验证操作，比如当一个类型的二进制信息被加载进 JVM 之前，就必须做一些特定的验证，以免这些二进制信息对 JVM 虚拟机产生不良的影响，或者因此导致 JVM 进程崩溃。之前笔者曾经提及过，验证阶段 JVM 所执行的一系列验证操作大致可以划分为：格式验证、语义分析、操作验证，以及符号引用验证等，那么在类加载阶段，验证阶段中的格式验证操作将会与之一起执行，格式验证的主要任务就是检查当前正在加载的字节码文件是否符合 JVM 规范，是否是一个有效的字节码文件，如果不是一个有效的字节码文件 JVM 就将会抛出 java.lang.VerifyError 异常。格式验证主要检查字节码文件中的前四个字节是否为 0xCAFEBABE，笔者在 2.6 小节中曾经提及过，这 4 个字节所代表的含义就是 magic，简

单来说，magic 就是 JVM 用于校验所读取的目标文件是否是一个有效且合法的字节码文件；排列在 magic 后的第 5 个和第 6 个字节所代表的含义就是编译的次版本号，而第 7 个和第 8 个字节就是编译的主版本号，验证字节码文件的主版本号和次版本号同样也是格式验证的任务之一，因为如果是高版本的 JDK 编译的字节码文件，自然不能在低版本的 JVM 中运行，否则 JVM 会抛出 java.lang.UnsupportedClassVersionError 异常。在此大家需要注意，尽管这一部分的验证操作是在类加载阶段完成的，但它逻辑上仍然是属于验证阶段。

只有当成功通过格式验证之后，类加载器才会成功将类的二进制数据信息加载到方法区中。而后续的其他验证操作都直接在方法区中进行。接下来 JVM 就会开始执行下一个阶段的验证操作，而正式的验证操作首先会对元数据信息进行验证，而这个验证阶段的主要任务就是验证字节码信息是否符合 Java 语法规范，如下所示：

- 检查一个被标记为 final 的类型是否包含派生类；
- 检查一个类中的 final 方法是否被派生类进行重写；
- 确保超类与派生类之间没有不兼容的一些方法声明（比如方法签名相同，但方法的返回值不同）。

当成功执行完语义验证之后，JVM 接下来就会进入操作验证阶段。在这个验证阶段中，JVM 会对类型的方法执行验证，以确保一个类的方法在执行时，不会对 JVM 产生不良的影响，不会因此导致 JVM 的进程出现崩溃。就像在 8.1.2 节中提及的那样，在操作数栈中的数据必须进行正确的操作，比如不能在入栈 2 个 int 类型的数值后，却把它们当做 long 类型的数值去操作，或者入栈 2 个 double 类型的数值后，使用 iadd 指令对它们执行加法运算等情况出现。

在验证阶段的最后一个验证操作中，则主要是对常量池中的各种符号引用执行验证，因此这一个阶段也就称之为符号引用验证。在此大家需要注意，由于在 JVM 的具体实现中，链接阶段中的解析操作往往发生在初始化之后，因此这一个阶段的验证则会在解析阶段中才会执行。简单来说，解析阶段的主要任务就是将常量池中所有的符号引用全部转换为直接引用，那么符号引用验证的主要任务就是验证需要被转换为直接引用的这些符号引用是否正确，比如是否能够通过符号引用中通过字符串描述的全限定名定位到指定的类型上，或者是符号引用中的类成员信息的访问修饰符是否能够被当前类执行访问操作等。

当 JVM 执行完验证阶段后，就将会执行到连接阶段中的下一个准备阶段中。

7.2.3 准备阶段

当成功经历过验证阶段后，JVM 接下来要做的事情就是对存放在方法区中类数据信息的类变量执行初始化，这里所执行的初始化操作并非是指类加载阶段中的初始化操作，这里

仅仅只是为类中的所有静态变量分配内存空间，并为其设置一个初始值（由于还没有产生对象，因此实例变量将不在此操作范围内），而非用户手动执行赋值操作。如下所示：

表 7-2 原始类型与引用类型的初始值

变量类型	初始值
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d
char	‘\u0000’
boolean	False
reference	null

在此大家需要注意，JVM 实现其实并不支持 boolean 类型，因此在 JVM 内部，boolean 类型往往被实现为一个 int 类型，初始值为 0 也就代表着 false。当然 boolean 类型的变量，尽管在 JVM 中式当做 int 类型来实现，但在初始化时也总会被初始化为一个 false 值。

7.2.4 解析阶段

当成功经历过链接阶段中的验证阶段和准备阶段之后，就可以进入链接中的最后一个解析操作阶段。之前也说过，由于 Java 虚拟机规范并没有明确要求解析阶段一定要按照顺序执行，因此解析阶段可以等到初始化之后再执行。简单来说，解析阶段的主要任务就是将字节码常量池中的符号引用全部转换为直接引用，包括类、接口、方法和字段的符号引用。

7.2.5 初始化阶段

类加载过程中的最后一个阶段就是初始化，在这个阶段中，JVM 会将一个类中所有被 static 关键字标示的代码统统执行一遍，如果执行的是静态变量，那么就会使用用户指定的值覆盖掉之前在准备阶段中 JVM 为其设置的初始值，当然如果程序中并没有为静态变量显式指定赋值操作，那么所持有的值仍然是之前的初始值；反之如果执行的是 static 代码块，那么在初始化阶段中，JVM 就将会执行 static 代码块中定义的所有操作。

在 Java 程序中，开发人员可以通过使用初始化语句或者静态初始化语句为一个类中的类变量执行初始化操作。如下所示：

代码 7-9 初始化类变量

```
/**
 * 使用初始化语句或静态代码块初始化类变量
 *
 * @author JohnGao
 */
public class InitTest {
    public static int value1 = 100;
    public static int value2;
    static {
        value2 = value1;
    }
}
```

所有的类变量初始化语句和静态代码块都会在 Java 源码执行字节码编译时，被前端编译器放在收集器里，存放到一个特殊的方法中，这个方法就是<clinit>()方法。对于类来说，这个方法可以被称之为类初始化方法，而对于接口来说，则可以称之为接口初始化方法。简单来说，<clinit>()方法的作用就是初始化一个类中的类变量，使用用户指定的值覆盖掉之前在准备阶段中 JVM 为其设置的初始值。

既然<clinit>()方法在编译成字节码的时候生成，那么我们就可以通过命令“javap”来查看编译后的字节码文件。如下所示：

代码 7-10 InitTest 类的字节码指令

```
public class InitTest {
  public static int value1;
  public static int value2;
  public InitTest();
    Code:
       0: aload_0
       1: invokespecial #1        // Method java/lang/Object."<init>":()V
       4: return
  static {};
    Code:
       0: bipush        100
       2: putstatic     #2          // Field value1:I
       5: getstatic     #2          // Field value1:I
       8: putstatic     #3          // Field value2:I
      11: return
}
```

或许大家会对上述的反编译结果产生疑问，为什么生成的字节码文件中并没包含<clinit>()方法？其实上述字节码文件中 static{}就是<clinit>()方法，只是 javac 编译器将其名

字进行了更改而已。本书示例了包含在 sun.tools.java 目录下的 Constants 和 MemberDefinition 类的代码片段，如下所示：

代码 7-11　编译字节码时更改<clinit>()方法的名称

```
package sun.tools.java;

public
interface Constants extends RuntimeConstants {
    // ...
    Identifier idClassInit = Identifier.lookup("<clinit>");
    // ...
}

package sun.tools.java;

// ...

public
class MemberDefinition implements Constants {
    // ...
    public final boolean isInitializer() {
        return getName().equals(idClassInit);
    }

    public String toString() {
        if (isInitializer()) {
            return isStatic() ? "static {}" : "instance {}";
        } else // ...
    }
}
```

并且任何 invoke 之类的字节码指令也无法调用<clinit>()方法，因为该方法只能在类加载的过程中被 JVM 所调用。在上述代码示例中，<clinit>()方法首先会执行 InitTest 类中的类变量 value1 的初始化操作，当 value1 初始化完成后，接下来才会进入静态代码块中对类变量 value2 执行初始化操作。也就是说类变量的初始化顺序与定义在代码中的类变量的顺序有关，因此上述程序中类变量初始化语句会优先在静态代码块之前执行。

在此大家需要注意，如果一个类拥有直接超类的话，如果超类还没有被执行初始化，那么优先对超类初始化，但是在<clinit>()方法内部却并不会显式调用超类的<clinit>()方法，会由 JVM 负责保证在一个类的<clinit>()方法执行之前，它的超类的<clinit>()方法已经被执行了。如果说一个类并没有任何显式的超类，那么 java.lang.Object 就会是这个类的直接超类，或者在有超类的情况下，java.lang.Object 同样也会是第一个被优先执行初始化的类型，然后才会初始化这个类在继承关系上的所有超类。

JVM 必须确保一个类在被初始化的过程中，如果是多个线程需要同时初始化它，仅仅只能允许其中的一个线程对其执行初始化操作，而其余的线程则必须等待，只有在活动线程执行完对类的初始化操作之后，才会通知正在等待的其他线程。

当然也并不是说所有的类型都需要在字节码文件中拥有一个<clinit>()方法，如果一个类中并没有声明任何的类变量，也没有静态代码块，那么这个类在编译为字节码后，字节码文件中将不会包含<clinit>()方法。当然如果一个类中声明类变量，但是没有明确使用类变量的初始化语句以及静态代码块来执行初始化操作时，编译的字节码文件中同样不会包含<clinit>()方法。除此之外，如果一个类中包含静态 final 变量的类变量初始化语句，而且这些类变量初始化语句采用编译时常量表达式，在编译后的字节码文件中同样也不会包含<clinit>()方法，也就是说，只有那些的确需要执行 Java 代码来为类变量执行赋值操作的类型在编译之后才会在字节码文件中生成<clinit>()方法。

7.3 实战：字节码文件的加密与解密

Java 源文件的编译结果缺省为字节码文件，也就是后缀名为“.class”的文件，那么在很多情况下，我们并不希望看到编译后的字节码文件被其他人使用一些反编译工具将程序中的字节码文件反编译为源文件以泄露一些非开源的数据信息，尤其是随着目前网络的日益普及，互联网的飞速发展，各式各样的 Java 反编译工具日新月异地更新着。那么尽管不能百分百有效组织编译后的字节码文件被反编译，那么至少可以通过一些字节码加解密技术来增加字节码被反编译的难度。尽管目前的加密算法非常多，但考虑到笔者在实际的项目中大部分情况下均是使用 3DES 对称加密算法执行项目中一些请求处理的加解密操作，那么在本章的实战小节中，笔者将会使用 3DES 对称加密算法对编译后的字节码文件进行加密处理，并在执行加载的时候配合笔者编写的自定义类加载器来重新定义类的加载规则，以便在自定义类加载器中实现一些自定义的处理逻辑，比如对加密后的字节码文件执行解密操作等。

在正式开始对编译后的字节码文件执行加解密操作之前，我们首先来认识一下什么是 3DES 算法。3DES 也可以称之为 Triple DES，它是 DES 加密算法的一种模式，使用 3 条 56 位的秘钥对数据进行三次加密。简单来说，由于计算机运算能力的增强，原版 DES 密码的密钥长度变得容易且被暴力破解，3DES 即是设计用来提供一种相对简单的方法，通过增加 DES 的密钥长度来避免类似的攻击，而不是设计一种全新的块密码算法，也就是说 3DES 相对于 DES 来说更加安全。3DES 的加解密过程如下：

- 加密过程为：C=Ek3(Dk2(Ek1(P)))。
- 解密过程为：P=Dk1(EK2(Dk3(C)))。

Ek()和 Dk()代表 DES 算法的加密和解密过程，K 代表 DES 算法使用的密钥，P 代表明文，C 代表密文。当大家对 3DES 算法有所了解后，接下来我们首先来看看，在程序中，如何利用 Java API 对 3DES 算法提供的支持来对目标数据执行加解密操作。如下所示：

代码 7-12　使用 3DES 算法执行数据加解密操作

```
/**
 * 使用 3DES 算法对目标数据执行加解密操作
 *
 * @author JohnGao
 */
public class Use3DES {
    /* 定义需要使用的加解密算法 */
    private static final String ALGORITHM = "DESede"; // 定义加密算法,
可用

    /**
     * 使用 3DES 算法对目标数据执行加密操作
     *
     * @author JohnGao
     *
     * @param key
     *            192 位的加密 key
     *
     * @param src
     *            需要执行加密的数据
     *
     * @exception Exception
     *
     * @return byte[] 执行加密后的数据
     */
    public static byte[] encrypt(byte[] key, byte[] src) {
        byte[] value = null;
        try {
            /* 生成秘钥 key */
            SecretKey deskey = new SecretKeySpec(key, ALGORITHM);
            /* 对目标数据执行加密操作 */
            Cipher cipher = Cipher.getInstance(ALGORITHM);
            cipher.init(Cipher.ENCRYPT_MODE, deskey);
            value = cipher.doFinal(src);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return value;
    }

    /**
```

```
     * 使用 3DES 算法对目标数据执行解密操作
     *
     * @author JohnGao
     *
     * @param key
     *            192 位的加密 key
     *
     * @param src
     *            需要执行解密的数据
     *
     * @exception Exception
     *
     * @return byte[] 执行解密后的数据
     */
    public static byte[] decrypt(byte[] key, byte[] src) {
        byte[] value = null;
        try {
            /* 生成秘钥 key */
            SecretKey deskey = new SecretKeySpec(key, ALGORITHM);
            /* 对目标数据执行解密操作 */
            Cipher cipher = Cipher.getInstance(ALGORITHM);
            cipher.init(Cipher.DECRYPT_MODE, deskey);
            value = cipher.doFinal(src);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return value;
    }

    @Test
    public void run() {
        try {
            byte[] key = "012345678999876543210abcd".getBytes();
            byte[] encoded = encrypt(key,
                    "测试数据...".getBytes("utf-8"));
            System.out.println("加密后的数据->" + new String(encoded));
            System.out.println("解密后的数据->"
                    + new String(decrypt(key, encoded), "utf-8"));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

程序输出如下：

```
加密后的数据->P 涪-↑9  浪 k0 肪
解密后的数据->测试数据...
```

上述代码示例中，encrypt()方法中的任务就是使用 3DES 算法对目标数据执行加密操作，由于 3DES 是对称加解密，所以在执行加密之前的第一步就是通过 SecretKeySpec 生成一个秘钥 key，然后在 Cipher 执行 init()方法时，指定模式为 Cipher.ENCRYPT_MODE，也就是当前所需要执行的操作为加密操作，最后 doFinal()方法将会对目标数据源按照指定的规则生成加密后的数据返回。解密操作类似于加密操作，只是在 decrypt()方法中需要指定 Cipher 执行 init()方法的执行模式为 Cipher.DECRYPT_MODE。

当大家知道究竟应该在 Java 语法层面使用 API 提供的方法来执行 3DES 算法的加解密操作后，接下来我们要做的事情就是对编译后的字节码文件执行 3DES 加密，再使用自定义类加载器将加密后的字节码文件先解密，再加载。如下所示：

代码 7-13　编写自定义类加载器对加密后的字节码文件执行解密

```
/**
 * 自定义类加载器
 *
 * @author JohnGao
 */
public class MyClassLoader extends ClassLoader {
    private String byteCode_Path;
    private byte[] key;

    public MyClassLoader(String byteCode_Path, byte[] key) {
        this.byteCode_Path = byteCode_Path;
        this.key = key;
    }

    @Override
    protected Class<?> findClass(String className)
            throws ClassNotFoundException {
        byte value[] = null;
        BufferedInputStream in = null;
        try {
            in = new BufferedInputStream(new FileInputStream
                    (byteCode_Path + className + ".class"));
            value = new byte[in.available()];
            in.read(value);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            /* 释放资源 */
            if (null != in) {
                try {
                    in.close();
                } catch (IOException e) {
                    e.printStackTrace();
```

```
                    }
                }
            }
            /* 对加加密后的字节码文件执行解密操作 */
            value = Use3DES.decrypt(key, value);
            /* 将 byte 数组转换为一个类的 Class 对象实例 */
            return defineClass(value, 0, value.length);
        }

        public static void main(String[] args) {
            BufferedInputStream in = null;
            try {
                in = new BufferedInputStream(new FileInputStream(
                        "D:/TestDemo.class"));
                byte[] src = new byte[in.available()];
                in.read(src);
                in.close();
                /* 对称加解密 key */
                byte[] key = "0123456789876543210abcd".getBytes();
                /* 先对字节码文件执行加密操作 */
                BufferedOutputStream out = new BufferedOutputStream(
                        new FileOutputStream("D:/test/TestDemo.class"));
                out.write(Use3DES.encrypt(key, src));
                out.close();
                MyClassLoader classLoader = new MyClassLoader("D:/test/",
key);
                System.out.println(classLoader.loadClass("TestDemo")
                        .getClassLoader().getClass().getName());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

上述代码示例中，笔者首先对编译后的字节码文件执行了加密操作，在重新保存之后再使用笔者自己编写的自定义类加载器对加密后的字节码文件进行解密后执行加载操作。

7.4 本章小结

本章笔者详细地讲解了类的加载机制必须经过类加载、连接和初始化 3 个主要阶段，其中类加载过程中，笔者对类加载器的相关知识点进行了深入讲解，包括如何在程序中编写一个自定义类加载器来重新定义类的加载机制以及如何实现一些自定义的处理逻辑。当大家清楚类加载机制后，本书的下一章将会着重讲解 JVM 中的执行引擎究竟是如何执行字节码的。

第 8 章

剖析HotSpot的架构模型与执行引擎

对于计算机而言，它能够做到的仅仅只是识别本地机器指令，对于那些使用编程语言编写的源代码则首先需将其编译为对应平台的本地机器指令之后计算机才能够进行正常识别和执行。随着时间的逐步推移，编程语言正在一步一步地朝着人类的思维方式日新月异的进化着，在面向对象等高级语言层出不穷的今天，诸如 Java 之类的编程语言的源代码编译结果并非还是本地机器指令，而是中间代码。这意味着编译结果一旦不依赖于特定的平台后，完全可以做到体系结构中立，一次编译处处运行（Write Once，Run Anywhere）。

既然 Java 代码的编译结果是字节码，那么这就需要一种运行介质能够让其高效运行起来，毕竟计算机并不能够直接别这些中间代码。HotSpot VM 是目前市面上高性能虚拟机的代表作之一。它采用解释器与即时编译器并存的架构，当虚拟机启动的时候，解释器可以首先发挥作用，而不必等待即时编译器全部编译完成再执行，这样可以省去许多不必要的编译时间。并且随着程序运行时间的推移，即时编译器逐渐发挥作用，根据热点探测功能，将有价值的字节码编译为本地机器指令，以换取更高的程序执行效率。在今天，Java 程序的运行性能早已脱胎换骨，已经达到了可以和 C/C++程序一较高下的地步，并且 Java 技术自身的诸多优势同样也是 C/C++无法比拟的。

8.1 栈帧的组成结构

在 Java 虚拟机规范中，Java 栈（Java Stack）也可以被称之为 Java 虚拟机栈（Java Virtual Machine Stack），它同 PC 寄存器一样都是线程私有的，并且生命周期与线程的生命周期保持一致。Java 栈主要用于存储栈帧（Stack Frame），而栈帧中则负责存储局部变量表、操作

数栈、动态链接和方法返回值等信息。

在面向对象（Object Oriented）的世界中，类（Class）与对象（Object）是最基本的概念，字段和方法是一个类的主要构成元素。当实例化对象后，字段便可以理解为一个对象的各种“器官”，而方法则可以被理解为一个对象的一系列“行为”，因此方法就是程序用于执行命令的关键，那么方法和栈帧之间又存在什么样的关系呢？简单来说，栈帧是一种用于支持 JVM 调用/执行程序方法的数据结构，它是方法的执行环境，每一个方法被调用时都会创建一个独立的栈帧以便维系所需的各种数据信息，栈帧伴随着方法的调用而创建，伴随着方法的执行结束而销毁，那么每一个方法从调用到执行结束的过程，就对应着 Java 栈中一个栈帧从入栈到出栈的过程，并且无论方法的调用状态是否正常都算作方法结束。在此大家需要注意，不同线程中所包含的栈帧是不允许存在相互引用的。

在栈帧中，局部变量表和操作数栈所需的容量大小在编译期就可以完全被确定下来，并保存在方法的 Code 属性中，也就是说，栈帧究竟需要分配多大的内存空间完全取决于具体的 JVM 实现和方法调用时分配的实际内存。在一条活动线程中，只有当前正在执行的方法的栈帧（栈顶栈帧）是有效的，这个栈帧也被称之为当前栈帧（Current Frame），与当前栈帧相对应的方法就是当前方法（Current Method），定义这个方法的类就是当前类（Current Class）。如图 8-1 所示。

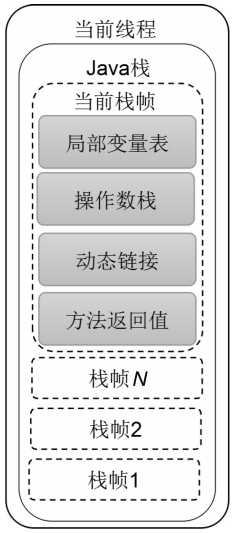


图 8-1 栈帧的组成结构

大家思考一下，既然一个线程中只有当前正在执行的方法的栈帧才是当前栈帧，那么如果当前方法在执行过程中调用了另外一个新的方法时，当前栈帧会发生变化吗？如下所示：

代码 8-1 当前栈帧示例

```
/**
 * 谁是当前栈帧?
 *
 * @author JohnGao
 */
public class CurrentFrameTest {
  public void methodA() {
      System.out.println("当前栈帧对应的方法->methodA");
      methodB();
      System.out.println("当前栈帧对应的方法->methodA");
  }
  public void methodB() {
      System.out.println("当前栈帧对应的方法->methodB");
  }
}
```

在上述程序示例中，如果与 methodA()方法相对应的栈帧是当前栈帧，那么当 methodA()方法内部调用了 methodB()方法时，则会有一个与 methodB()方法相对应的新栈帧作为当前栈帧被创建，也就是说，程序的控制权将会移交给 methodB()方法。不过当 methodB()方法执行完成并返回后，当前栈帧随之被丢弃，前一个栈帧又重新变为当前栈帧。

8.1.1 局部变量表

局部变量表（Local Variables Table）也可以称之为本地变量表，它包含在一个独立的栈帧中。顾名思义，局部变量表主要用于存储方法参数和定义在方法体内的局部变量，这些数据类型包括各类原始数据类型、对象引用（reference），以及 returnAddress 类型。局部变量表所需的容量大小在编译期就可以被完全确定下来，并保存在方法的 Code 属性中。大家思考一下，既然方法体内定义的局部变量是存储在栈帧中的局部变量表里的，那么原始数据类型的成员变量的值是否也存储在局部变量表中呢？其实如果是定义在方法体外的成员变量，不止是作用域发生了变化，更重要的是，其值也并非还是存储在局部变量表里，而是存储在对象内存空间的实例数据中，整体来看即存储在 Java 堆区内。简单来说，与线程上下文相关的数据存储在 Java 栈中，反之则存储在 Java 堆区内。

局部变量表可以看做是专门用于存储局部变量值的一种类似于线性表（Linear List）的数据结构。参考《Java 虚拟机规范 Java SE7 版》的描述来看，局部变量表中最小的存储单元是 Slot（变量槽），一个 Slot 可以存储一个类型为 boolean、byte、char、short、float、reference 以及 returnAddress 小于或等于 32bit 的数值，2 个 Slot 可以存储一个类型为 long 或 double 的 64bit 数值。JVM 会为局部变量表中的每一个 Slot 都分配一个访问索引，通过这个索引即可成功访问到局部变量表中指定的局部变量值，访问索引从 0 开始到小于局部变

量表最大的 Slot 长度，如图 8-2 所示。在此大家需要注意，由于 long 和 double 类型的二进制位数是 64bit，那么当使用这 2 个类型存储数据时，理论上占用的是 2 个连续的 Slot，如果需要访问局部变量表中一个 64bit 的局部变量值时，只需要使用前一个索引即可。这就好比一个 double 类型的值存储在局部变量表中其 Slot 的访问索引为 *n*，当我们需要取出这个局部变量值时，只需要根据索引 *n* 便可以成功取出 *n* 和 *n*+1 的值，也就是一个完整的 64bit 的数据值。当然关于是否一定需要使用 2 个连续的 Slot 来存储一个 64bit 的值，Java 虚拟机规范其实并没有明确要求，这主要还需要根据 JVM 的具体实现而定。除此之外，一个 Slot 究竟应该占用多大的内存空间 Java 虚拟机规范同样也没有明确的要求，但最好使用 32bit 的内存空间用于存储 boolean、byte、char、short、float、reference 及 returnAddress 等类型的值，当然这并不会意味着 Slot 的内存大小就一定会固定为 32bit，因为 Slot 的内存大小允许根据处理器、操作系统或 JVM 实现的不同而产生相应的变化。

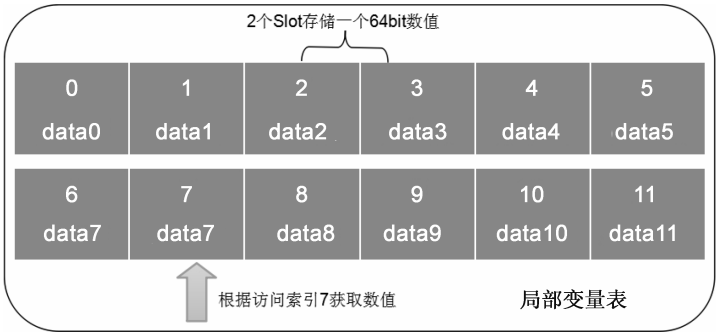


图 8-2 根据索引访问局部变量表中的数据

JVM 使用局部变量表来完成方法调用时的参数传递，当一个实例方法被调用的时候，它的方法参数和方法体内部定义的局部变量将会按照顺序被复制到局部变量表中的每一个 Slot 上。访问索引为 0 的 Slot 一定存储的是与被调用实例方法相对应的对象引用（通过 Java 语法层面的“this”关键字便可访问到这个参数），而后续的其他方法参数和方法体内定义的成员变量则会按照顺序从局部变量表中索引为 1 的 Slot 位置处展开复制。

8.1.2 操作数栈

每一个独立的栈帧中除了包含局部变量表以外，还包含一个后进先出（Last-In-First-Out）的操作数栈，也可以称之为表达式栈（Expression Stack）。操作数栈和局部变量表在访问方式上存在着较大差异，操作数栈并非采用访问索引的方式来进行数据访问的，而是通过标准的入栈和出栈操作来完成一次数据访问。每一个操作数栈都会拥有一个明确的栈深度用于存储数值，一个 32bit 的数值可以用一个单位的栈深度来存储，而 2 个单位的栈深度则可以保存一个 64bit 的数值，当然操作数栈所需的容量大小在编译期就可以被完全确定下来，并保

存在方法的 Code 属性中。

在 HotSpot 中，除了 PC 寄存器之外，再也没有包含其他任何的寄存器，并且之前曾经提及过，HotSpot 中任何的操作都需要经过入栈和出栈来完成，那么由此可见，HotSpot 的执行引擎架构必然就是基于栈式架构，而非传统的寄存器架构。简单来说，操作数栈就是 JVM 执行引擎的一个工作区，当一个方法被调用的时候，一个新的栈帧也会随之被创建出来，但这个时候栈帧中的操作数栈却是空的，只有方法在执行的过程中，才会有各种各样的字节码指令往操作数栈中执行入栈和出栈操作。比如在一个方法内部需要执行一个简单的加法运算时，首先需要从操作数栈中将需要执行运算的两个数值出栈，待运算执行完成后，再将运算结果入栈。如下所示：

代码 8-2　执行加法运算的字节码指令

```
public void testAddOperation();
    Code:
      0: bipush        15
 2: istore_1
 3: bipush        8
 5: istore_2
 6: iload_1
 7: iload_2
 8: iadd
 9: istore_3
 10: return
```

在上述字节码指令示例中，首先会由“bipush”指令将数值 15 从 byte 类型转换为 int 类型后压入操作数栈的栈顶（对于 byte、short 和 char 类型的值在入栈之前，会被转换为 int 类型），当成功入栈之后，“istore_1”指令便会负责将栈顶元素出栈并存储在局部变量表中访问索引为 1 的 Slot 上。接下来再次执行“bipush”指令将数值 8 压入栈顶后，通过“istore_2”指令将栈顶元素出栈并存储在局部变量表中访问索引为 2 的 Slot 上。“iload_1”和“iload_2”指令会负责将局部变量表中访问索引为 1 和 2 的 Slot 上的数值 15 和 8 重新压入操作数栈的栈顶，紧接着“iadd”指令便会将这 2 个数值出栈执行加法运算后再将运算结果重新压入栈顶，“istore_3”指令会将运算结果出栈并存储在局部变量表中访问索引为 3 的 Slot 上。最后“return”指令的作用就是方法执行完成之后的返回操作。在操作数栈中，一项运算通常由多个子运算（subcomputation）嵌套进行，一个子运算过程的结果可以被其他外围运算所使用。

在此大家需要注意，在操作数栈中的数据必须进行正确的操作。比如不能在入栈 2 个 int 类型的数值后，却把它们当做 long 类型的数值去操作，或者入栈 2 个 double 类型的数值后，使用 iadd 指令对它们执行加法运算等情况出现。

8.1.3 动态链接

每一个栈帧内部除了包含局部变量表和操作数栈之外，还包含一个指向运行时常量池中该栈帧所属方法的引用，包含这个引用的目的就是为了支持当前方法的代码能够实现动态链接（Dynamic Linking）。在 6.2.3 节中，笔者曾经提及过运行时常量池，一个有效的字节码文件中除了包含类的版本信息、字段、方法以及接口等描述信息外，还包含一项信息，那就是常量池表（Constant Pool Table），那么运行时常量池就是字节码文件中常量池表的运行时表示形式。在一个字节码文件中，描述一个方法调用了另外的其他方法时，就是通过常量池中指向方法的符号引用（Symbolic Reference）来表示的，那么动态链接的作用就是为了将这些符号引用转换为调用方法的直接引用。

在 JVM 中，将符号引用转换为调用方法的直接引用与方法的绑定机制相关。当一个字节码文件被装载进 JVM 内部时，如果被调用的目标方法在编译期可知，且运行期保持不变时，那么在这种情况下将调用方法的符号引用转换为直接引用的过程称之为静态链接。相反如果被调用的方法在编译期无法被确定下来，也就是说，只能够在程序运行期将调用方法的符号引用转换为直接引用，由于这种引用转换过程具备动态性，因此也就被称之为动态链接。既然引用的转换方式与方法的绑定机制相关，那么究竟什么是方法绑定呢？在 Java 中一共拥有两种方法绑定方式，分别是早期绑定（Early Binding）和晚期绑定（Late Binding）。顾名思义，早期绑定就是指被调用的目标方法如果在编译器可知，且运行期保持不变时，即可将这个方法与所属的类型进行绑定，这样一来，由于明确了被调用的目标方法究竟是哪一个，因此也就可以使用静态链接的方式将符号引用转换为直接引用。相反如果被调用的方法在编译期无法被确定下来，只能够在程序运行期根据实际的类型绑定相关的方法，这种绑定方式也就被称之为晚期绑定。在早期基于过程式的编程语言中，由于本身并不具备面向对象的一些特性，自然也就无法在语法层面上使用多态（Polymorphism）特性来实现方法的重写（Override）操作，也就是说，基于过程式的编程语言从严格意义上来说仅仅只存在一种绑定方式，那就是早期绑定。但随着高级语言的横空出世，类似于 Java 一样的基于面向对象的编程语言如今越来越多，尽管这类编程语言在语法风格上存在一定的差别，但是它们彼此之间始终保持着一个共性，那就是都支持封装、继承和多态等面向对象特性，既然这一类的编程语言具备多态特性，那么自然也就具备早期绑定和晚期绑定两种绑定方式。

在 Java 中，开发人员并不需要在程序中显式指定某一个方法需要在运行期支持晚期绑定，因为除了 final 方法外，几乎所有的方法都是默认基于晚期绑定的。如下所示：

代码 8-3 晚期绑定示例

```
/**
 * 晚期绑定(Late Binding)示例
```

```
 *
 * @author JohnGao
 */
public class LateBinding {
    public static void getName(Animal animal) {
        animal.name();
    }

    public static void main(String[] args) {
        getName(new Tiger());
        getName(new Pig());
    }
}

interface Animal {
    public void name();
}

class Tiger implements Animal {
    @Override
    public void name() {
        System.out.println("我是 Tiger，我派生于 Animal");
    }
}

class Pig implements Animal {
    @Override
    public void name() {
        System.out.println("我是 Pig，我派生于 Animal");
    }
}
```

在上述程序示例中，接口 Animal 包含 Tiger 和 Pig 两个派生类，并且这两个派生类还重写了它的 name()方法。由于在编译期并不明确 LateBinding 类中的 getName()方法究竟需要调用哪一个 name()方法，也就无法使用静态链接的方式将符号引用转换为直接引用，因此这一类的方法就是基于晚期绑定的虚函数，其实虚函数的存在就是为了支持多态特性。与动态链接相反，如果程序中能够使用静态链接的方式将符号引用转换为直接引用的话，这一类的方法就是基于早期绑定的非虚函数。在此大家需要注意，从严格意义上来说，在 Java 中其实并不存在虚函数的概念，因为开发人员并不需要显式使用任何关键字去标示 Java 中的一个虚函数。简单来说，Java 中任何一个普通的方法其实都具备虚函数的特征，它们相当于 C++语言中的虚函数（C++中则需要使用关键字 virtual 来显式定义）。如果在程序中不希望某个方法拥有虚函数的特征时，则可以使用关键字 final 来标记这个方法。

8.1.4 方法返回值

一个方法在执行的过程中将会产生两种调用结果：一种是方法正常调用完成，而另外一种则是方法异常调用完成。如果是方法正常调用完成，那么这就意味着，被调用的当前方法在执行的过程中将不会有任何的异常被抛出，并且方法在执行的过程中一旦遇见字节码返回指令时，将会把方法的返回值返回给它的调用者，不过一个方法在正常调用完成之后究竟需要使用哪一个返回指令还需要根据方法返回值的实际数据类型而定。在字节码指令中，返回指令包含 ireturn（当返回值是 boolean、byte、char、short 和 int 类型时使用）、lreturn、freturn、dreturn 以及 areturn，另外还有一个 return 指令供声明为 void 的方法、实例初始化方法、类和接口的初始化方法使用。

与方法正常调用完成相反的就是方法异常调用完成。方法异常调用完成意味着当前方法在执行的过程中可能会因为某些错误的指令导致 JVM 抛出了异常，并且这些异常在当前方法中没有办法进行处理，或者方法在执行的过程中遇见了 athrow 指令显式抛出的异常，并且在当前方法内部没有捕获这个异常。总之，如果一个方法在执行的过程中抛出了异常，那么这个方法在调用完成之后将不会再有任何的返回值返回给它的调用者。

无论当前方法的调用结果是正常还是异常，都需要在执行完成之后返回到之前被调用的位置上，那么这个时候当前栈帧就承担着恢复调用者状态的责任。之前曾经提及过，在方法内部调用了另外一个方法时，将会有一个与当前方法相对应的新栈帧被创建出来，当方法调用完成之后，当前栈帧随之被丢弃，前一个栈帧又重新变为了当前栈帧，而被调用的方法如果带有返回值的话，其返回值将会被压入当前栈帧的操作数栈中，并更新 PC 寄存器中下一条需要执行的字节码指令。

8.2 HotSpot中执行引擎的架构模型

时至今日，似乎可以毫不夸张地说，Java 虚拟机实际上是一个比 Java 语言本身更成功、更优秀甚至更伟大的产品。笔者在 1.4.2 节中曾经提及过与语言无关特性，像 JRuby、Rhino、PHP（IBM WebSphere sMash PHP(P8)、Caucho Quercus）和 Scala 之类的编程语言本身其实并不依赖于 Java 语言，而是依赖于 Java 虚拟机作为程序的宿主运行环境。那么由此可见，Java 技术的核心就是 Java 虚拟机（JVM，Java Virtual Machine），因为所有的 Java 程序都运行在 Java 虚拟机内部。Java 虚拟机之所以被称之为虚拟机，是因为它是由规范所定义出的一种抽象计算机，它的主要任务就是负责装载字节码到其内部，解释/编译为对应平台上的机器指令执行，而执行引擎就是负责执行这项任务的关键，它是整个 Java 虚拟机中最重要的组成结构之一。

当然 Java 虚拟机规范并没有明确要求执行引擎究竟应该采用解释器还是 JIT 编译器的实现方式来执行字节码指令，不过早在 Java1.0 版本的时候 Sun 公司提供了一款基于纯解释器实现的 Java 虚拟机（Sun Classic VM），但由于解释器的执行效率非常低下，所以到了 1998 年 Sun 公司发布 Java1.2 版本的时候，基于即时编译的 JIT 编译器出现了，带给了 Java 虚拟机在运行效率上质的飞跃。如今的 HotSpot VM 内部采用的是解释器与 JIT 编译器并存的方案共同执行字节码指令，当虚拟机启动的时候，解释器首先发挥作用，而不必等待编译器全部编译完成再执行，这样可以省去许多不必要的编译时间。并且随着程序运行时间的推移，JIT 编译器将会逐渐发挥它的作用，根据热点探测功能，将有价值的字节码指令直接编译为本地机器指令，以便换取更高的程序执行效率。

既然谈到了执行引擎，那就不得不提到 Java 虚拟机的架构模型。简单来说，基于寄存器架构的虚拟机，在性能上会占有优势，像 Google 公司专门为 Android 平台研发的 Dalvik 虚拟机就是基于寄存器架构的。相反如果是基于栈式架构的虚拟机在设计和实现上相对来说会更加简单，而且更适用于一些资源受限的系统（关于寄存器架构和栈式架构虚拟机的更多区别，请阅读 8.2.2 节）。大家回想一下，Java 语言的设计初衷是否就是为了能够在嵌入式设备中运行呢？由于嵌入式设备大都是一些资源受限的系统，因此当时 James Gosling 博士等人为 HotSpot VM 选择了基于栈的架构模型。大家思考一下，尽管嵌入式平台如今早就不是 Java 程序的主流运行平台了（准确来说应该是 HotSpot VM 的宿主环境已经不局限于嵌入式平台了），那么为什么不将架构更换为基于寄存器的架构呢？之前也说过了，由于栈式架构在设计和实现上会更加简单，所以在非资源受限的系统下选择栈式架构也未尝不可。从 Java 语法层面上来说，方法定义了对象的一系列“行为”，所以执行引擎本质上就是通过调用一个个的方法来执行命令。每当调用一个新方法的时候，一个与当前方法相对应的当前栈帧也就被随之创建出来，栈帧伴随着方法的调用而创建，伴随着方法的执行结束而销毁，那么每一个方法从调用到执行结束的过程，就对应着 Java 栈中一个栈帧从入栈到出栈的过程。

8.2.1 本地机器指令

在计算机世界中，只有机器指令能够在其内部执行，那么在正式开始讲解 HotSpot 中的执行引擎究竟如何执行字节码指令之前，我们首先来了解下什么是机器指令。不过大家不必担心，笔者并不打算列举一些枯燥乏味的操作系统原理等相关知识来阐述本节内容。

作为一门优秀的面向对象编程语言，Java 编译结果屏蔽了与底层操作系统和物理硬件相关的一些特性，使得开发人员尽可能地只需关注于自身业务。当然字节码仅仅只是一个实现跨平台的通用契约而已，它并不能够直接运行在操作系统之上，因为字节码指令并非等价于本地机器指令，它内部包含的仅仅只是一些能够被 JVM 所识别的字节码指令、符号表，以及其他辅助信息。那么如果想要让一个 Java 程序运行起来，执行引擎就需要将字节码指

令解释/编译为对应平台上的本地机器指令才可以，简单来说，JVM 中的执行引擎充当了将高级语言翻译为机器语言的译者。

那么究竟什么是机器指令呢？机器指令其实就是一种能够被 CPU 直接识别并执行的指令，它以二进制编码作为表示形式。机器指令通常由操作码和操作数两部分构成，操作码决定了指令需要执行什么样的功能，而操作数则指定了需要参与运算的操作数，以及从哪里获取操作数、将运算结果存储在哪个位置（寄存器还是栈中）等。由于机器指令与 CPU 紧密相关，所以不同种类的 CPU 所对应的机器指令也就不同，并且它们的指令系统往往差别很大。在此大家需要注意，由于在实际的开发过程中，大部分开发人员都是使用的基于面向对象的高级语言（比如 Java、Python、JRuby 等），并且从高级语言翻译为机器语言有专门的编译器会负责完成，开发人员并不需要在意由于不同的物理架构所带来的差别。

8.2.2 寄存器架构与栈式架构之间的区别

在 8.2 节中，笔者只是简单地介绍了关于寄存器架构和栈式架构之间的区别。当然仅凭栈式架构在设计和实现上更加简单这一个理由还不足以让 JVM 的设计者们动心，那么笔者接下来将会从方方面面来阐述这两种架构之间的区别，让大家更加深刻地理解基于栈式架构所带来的好处和栈式架构的优点。

指令集不同

其实寄存器架构和栈式架构之间最本质的区别还是二者之间使用的指令集不同，在 8.2.1 节中笔者曾为大家讲解过什么是本地机器指令，指令通常由操作码和操作数两部分构成，操作码决定了指令需要执行什么样的功能，而操作数则指定了需要参与运算的操作数，以及从哪里获取操作数、将运算结果存储在哪个位置（寄存器还是栈中）等。那么根据指令操作方式的不同，我们便可以将指令划分为零地址指令、一地址指令、二地址指令和三地址指令等 *n* 地址指令，在此大家需要注意，由于 n 是一个自然数，因此也就意味着指令集可以是任意的 *n* 地址。在大部分情况下，基于寄存器架构的指令集往往都以一地址指令、二地址指令和三地址指为主，而基于栈式架构的指令集却是以零地址指令为主。

那么不同的地址指令之间到底存在什么区别呢？以三地址指令为例，在一个简单的二元运算操作中，三地址指令正好可以指定 2 个数据源和 1 个存储目标，这样必然能够非常灵活地将二元运算和赋值操作组合在一起。如下所示：

代码 8-4 三地址指令的表示形式

```
op dest, src1, src2
```

很明显，如果是使用三地址指令去执行一项二元运算操作的确非常灵活。相反如果使用

基于栈式架构的零地址指令去执行一项二元运算时，又会如何呢？就以代码 8-2 为例，其中的“iadd”指令并没有任何参数，甚至连数据源都没有办法进行指定，那么零指令地址究竟有什么用呢？其实零地址指令意味着数据源和存储目标都是隐含参数，其实现就是依赖于一种被称之为栈的数据结构。首先会由“bipush”指令将数值 15 从 byte 类型转换为 int 类型后压入操作数栈的栈顶（对于 byte、short 和 char 类型的值在入栈之前，会被转换为 int 类型），当成功入栈之后，“istore_1”指令便会负责将栈顶元素出栈并存储在局部变量表中访问索引为 1 的 Slot 上。接下来再次执行“bipush”指令将数值 8 压入栈顶后，通过“istore_2”指令将栈顶元素出栈并存储在局部变量表中访问索引为 2 的 Slot 上。“iload_1”和“iload_2”指令会负责将局部变量表中访问索引为 1 和 2 的 Slot 上的数值 15 和 8 重新压入操作数栈的栈顶，紧接着“iadd”指令便会将这 2 个数值出栈执行加法运算后再将运算结果重新压入栈顶，“istore_3”指令会将运算结果出栈并存储在局部变量表中访问索引为 3 的 Slot 上。

既然大家都已经知道字节码文件中的指令集设计就是基于零地址指令的，那么使用零地址指令会有什么好处呢？其实零地址指令相比其他形式的指令会显得更加紧凑，因为在一个字节码文件中，除了处理 2 个表跳转的指令外，其他都是按照 8 位字节进行对齐的，操作码可以只占一个字节大小，这就意味着将会有更多的空间用于存储其他指令。因此在空间紧缺的环境中，使用零地址指令的设计将会是不错的选择。当然有利就必然会有弊，零地址指令尽管拥有良好的紧凑性，但是它完成一个操作却往往需要比二地址指令或三地址指令花费更多的出栈和入栈指令。比如在 x86 平台中的 CPU 指令集就是基于二地址指令的，如果完成一项类似的运算操作，二地址指令只需花费 2 条指令即可。

基于栈式架构的优点

- □ 设计和实现更简单，适用于资源受限的系统；
- □ 避开了寄存器的分配难题；
- □ 指令集更加紧凑。

如果虚拟机选用了基于栈的架构，不仅在架构的设计和实现上会更加简单，而且更适用于资源受限的系统。所谓资源受限，通常情况下大都是指一些 CPU 运算效率低下、内存较小的嵌入式设备（比如机顶盒、打印机等）。如果大家细心的话，在阅读本章的前面几个小节中的内容后应该会对基于栈式架构的指令执行方式非常熟悉了。简单来说，每当调用一个新方法的时候，一个与当前方法相对应的当前栈帧也就随之被创建出来，栈帧伴随着方法的调用而创建，伴随着方法的执行结束而销毁，那么每一个方法从调用到执行结束的过程，就对应着 Java 栈中一个栈帧从入栈到出栈的过程。并且如果是方法内部需要执行运算时，无非就是对操作数栈中的栈顶元素频繁地执行入栈和出栈操作而已。

由于基于栈式架构的零地址指令的执行方式仅仅只是对栈顶元素操作，所以在设计上根

本就不需要考虑寄存器的分配问题，因此大幅度简化了虚拟机在架构设计上的复杂度。我们都知道字节码是 Java 程序实现跨平台运行的基石，但最终字节码指令仍然需要被装载进 JVM 内部由执行引擎负责将其解释/编译为对应平台上的机器指令执行，因此基于寄存器架构的 JVM 自然会丧失掉 Java 程序与生俱来的跨平台优势，这是因为在一些寄存器较少或是寄存器不规律的平台中（典型的 CISC 处理器的通用寄存器数量很少，例如 32 位的 x86 就只有 8 个 32 位通用寄存器（如果不算 EBP 和 ESP 那就是 6 个，现在一般都算上）；典型的 RISC 处理器的各种寄存器数量多一些，例如 ARM 有 16 个 32 位通用寄存器，Sun 的 SPARC 在一个寄存器窗口里则有 24 个通用寄存器（8 in，8 local，8 out)），仍然需要保证 Java 程序能够正常顺利地运行下去，这几乎是不现实的。在此大家需要注意，尽管 Google 公司研发的 Dalvik 是一款根据 ARM 平台而设计的基于寄存器架构实现的虚拟机，但是从严格意义上来说，Dalvik 却并没有完全按照 Java 虚拟机规范来进行设计和实现，并且它运行的也不是传统意义上的字节码文件，而是 Android 平台专属的 dex 文件。

除了设计和实现更加简单，以及避开了寄存器的分配难题等优点外，基于栈式架构的虚拟机还有一个优点之前也已经提到过了，那就是前端编译器所生成的字节码指令相对来说更加紧凑。这是因为在基于零地址指令的字节码文件中，除了处理 2 个表跳转的指令外，其他都是按照 8 位字节进行对齐的，操作码可以只占一个字节大小，这就意味着将会有更多的空间用于存储其他指令。而基于寄存器架构的 Dalvik 虚拟机所执行的字节码指令内部却是采用 16 位双字节的方式进行设计的。

基于寄存器架构的优点

- 性能优秀和执行高效；
- 花费更少的指令去完成一项操作。

综合来看，基于栈式架构的零地址指令设计更适用于通用的虚拟机（比如HotSpot），但实际上基于寄存器架构的虚拟机性能却显得更加高效。引用RednaxelaFX在博文《虚拟机随谈一》[13]中的一段描述，尽管基于寄存器架构的虚拟机所使用的零地址指令更加紧凑，但是完成一项操作的时候必然需要花费更多的入栈和出栈指令，这同时也意味着将需要更多的指令分派（instruction dispatch）次数和内存读/写次数。由于访问内存是执行速度的一个重要瓶颈，二地址指令或三地址指令虽然每条指令占的空间较多，但总体来说可以用更少的指令去完成一项操作，指令分派与内存读/写次数相对来说也都更少。

① 博文地址：http://rednaxelafx.iteye.com/blog/492667。

8.2.3 基于栈式架构的设计

当大家都已经知道 HotSpot 虚拟机基于栈式架构后，笔者接下来为大家汇总叙述一下执行引擎在执行的过程中究竟是如何与其相关的一些组件协同运作的，当大家理解本节所阐述的内容后，相信会对执行引擎的架构设计有更清楚的认识和了解。

如图 8-3 所示，在 JVM 中，PC 寄存器和 Java 栈都同属线程私有的运行时内存区，并且它们的生命周期与线程的生命周期也是保持一致的。每当启动一个新线程的时候，JVM 就会为当前线程分配一个独立的 PC 寄存器和 Java 栈空间。那么当线程调用一个方法的时候，一个与当前方法相对应的当前栈帧也就随之被创建出来，并存储在当前线程的独立栈空间中，栈帧伴随着方法的调用而创建，伴随着方法的执行结束而销毁，那么每一个方法从调用到执行结束的过程，就对应着 Java 栈中一个栈帧从入栈到出栈的过程。栈帧主要由 4 部分构成，分别是局部变量表、操作数栈、动态链接和方法返回值，其中局部变量表主要用于存储方法参数和定义在方法体内部的局部变量，这些数据类型包括各类原始数据类型、对象引用（reference）以及 returnAddress 类型；而操作数栈则作为执行引擎的一个运行时工作区，用于存储方法在执行过程中的各种临时数据；当前方法在执行的过程中不排除会调用其它方法，因此栈帧中动态链接的任务就是负责将调用方法的符号引用转换为直接引用；一旦方法在执行的过程中遇见字节码返回指令时，便会将方法的返回值返回给它的调用者。

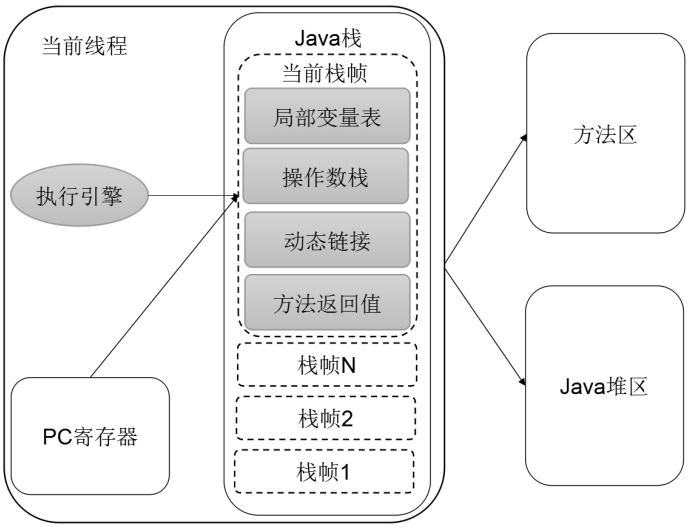


图 8-3 执行引擎的架构设计

在此大家需要注意，执行引擎在执行的过程中究竟需要执行什么样的字节码指令完全依

赖于 PC 寄存器，每当执行完一项指令操作后，PC 寄存器就会更新下一条需要被执行的指令地址。当然方法在执行的过程中，执行引擎有可能会通过存储在局部变量表中的对象引用准确定位到存储在 Java 堆区中的对象实例信息，以及通过对象头中的元数据指针定位到目标对象的类型信息。

8.2.4 调用call_stub()函数执行Java方法

在 2.3.7 节中，笔者曾经讲解过 Java 的启动函数 main()是如何被执行的。首先会由 Launcher 调用 CallStaticVoidMethod()函数，CallStaticVoidMethod()函数属于结构体 JNIEnv 类型的函数指针，该类型对应的目标函数为 jni_CallStaticVoidMethod()。在 jni_CallStaticVoidMethod()函数内部又调用了包含在/hotspot/src/share/vm/prims/jni.cpp 中的 jni_invoke_static()函数，而在 jni_invoke_static()函数内部，JavaCalls 首先会将从 GetStaticMethodId()函数中获取出一个全局唯一的方法 ID，然后将其转换为方法句柄后，再通过 call()函数最终执行 Java 程序的 main()方法。当然除了可以执行 Java 程序的 main()方法外，程序中所有的 Java 方法执行都需要依赖 JavaCalls 模块来完成调用，并由 JavaCalls 模块负责创建与当前方法相对应的当前栈帧。

在 JavaCalls 模块的实现中包含一个 call_helper()函数，该函数的作用就是调用 CallStub()函数执行 Java 方法，并返回调用方法的返回值给其调用者。在此大家需要注意，CallStub()函数只是一个函数指针，其对应的目标函数为 StubRoutines 模块中的 call_stub()函数。call_stub()函数的源代码包含在/hotspot/src/share/vm/runtime/javaCalls.cpp 中。

为了使大家阅读更方便，本书示例了 call_helper()函数的完整代码，如下所示：

代码 8-5 call_helper()函数的完整代码

```
    void JavaCalls::call_helper(JavaValue* result, methodHandle* m,
JavaCallArguments* args, TRAPS) {
      methodHandle method = *m;
      JavaThread* thread = (JavaThread*)THREAD;
      assert(thread->is_Java_thread(), "must be called by a java thread");
      assert(method.not_null(), "must have a method to call");
      assert(!SafepointSynchronize::is_at_safepoint(), "call to Java code
during VM operation");
      assert(!thread->handle_area()->no_handle_mark_active(),    "cannot
call out to Java here");

      CHECK_UNHANDLED_OOPS_ONLY(thread->clear_unhandled_oops();)

      // Verify the arguments

      if (CheckJNICalls)  {
```

```
        args->verify(method, result->get_type(), thread);
      }
      else debug_only(args->verify(method, result->get_type(), thread));

      // Ignore call if method is empty
      if (method->is_empty_method()) {
        assert(result->get_type() == T_VOID, "an empty method must return
a void value");
        return;
      }

    #ifdef ASSERT
      { klassOop holder = method->method_holder();
        // A klass might not be initialized since JavaCall's might be used
during the executing of
        // the <clinit>. For example, a Thread.start might start executing
on an object that is
        // not fully initialized! (bad Java programming style)
        assert(instanceKlass::cast(holder)->is_linked(), "rewritting must
have taken place");
      }
    #endif

      assert(!thread->is_Compiler_thread(), "cannot compile from the
compiler");
      if (CompilationPolicy::must_be_compiled(method)) {
        CompileBroker::compile_method(method, InvocationEntryBci,

CompilationPolicy::policy()->initial_compile_level(),
                        methodHandle(), 0, "must_be_compiled", CHECK);
      }

      // Since the call stub sets up like the interpreter we call the
from_interpreted_entry
      // so we can go compiled via a i2c. Otherwise initial entry method
will always
      // run interpreted.
      address entry_point = method->from_interpreted_entry();
      if (JvmtiExport::can_post_interpreter_events()
         && thread->is_interp_only_mode()) {
        entry_point = method->interpreter_entry();
      }

      // Figure out if the result value is an oop or not (Note: This is a
different value
      // than result_type. result_type will be T_INT of oops. (it is about
size)
      BasicType result_type = runtime_type_from(result);
```

```
      bool   oop_result_flag   =   (result->get_type()   ==   T_OBJECT   ||
result->get_type() == T_ARRAY);

      // NOTE: if we move the computation of the result_val_address inside
      // the call to call_stub, the optimizer produces wrong code.
      intptr_t*                    result_val_address                    =
(intptr_t*)(result->get_value_addr());

      // Find receiver
      Handle  receiver  =  (!method->is_static())  ?  args->receiver()  :
Handle();

      // When we reenter Java, we need to reenable the yellow zone which
      // might already be disabled when we are in VM.
      if (thread->stack_yellow_zone_disabled()) {
        thread->reguard_stack();
      }

      // Check that there are shadow pages available before changing thread
state
      // to Java
      if (!os::stack_shadow_pages_available(THREAD, method)) {
        // Throw stack overflow exception with preinitialized exception.
        Exceptions::throw_stack_overflow_exception(THREAD,      __FILE__,
__LINE__, method);
        return;
      } else {
        // Touch pages checked if the OS needs them to be touched to be mapped.
        os::bang_stack_shadow_pages();
      }

      // do call
      { JavaCallWrapper link(method, receiver, result, CHECK);
        { HandleMark hm(thread);  // HandleMark used by HandleMarkCleaner
          StubRoutines::call_stub()(
            (address)&link,
       // (intptr_t*)&(result->_value), // see NOTE above (compiler problem)
            result_val_address,      // see NOTE above (compiler problem)
            result_type,
            method(),
            entry_point,
            args->parameters(),
            args->size_of_parameters(),
            CHECK
          );

          result = link.result();  // circumvent MS C++ 5.0 compiler bug
(result is clobbered across call)
```

```
          // Preserve oop return value across possible gc points
          if (oop_result_flag) {
            thread->set_vm_result((oop) result->get_jobject());
          }
        }
      } // Exit JavaCallWrapper (can block - potential return oop must be
preserved)

      // Check if a thread stop or suspend should be executed
      // The following assert was not realistic.  Thread.stop can set that
bit at any moment.
      //assert(!thread->has_special_runtime_exit_condition(), "no async.
exceptions should be installed");

      // Restore possible oop return
      if (oop_result_flag) {
        result->set_jobject((jobject)thread->vm_result());
        thread->set_vm_result(NULL);
      }
    }
```

8.2.5 栈顶缓存（Top-of-Stack Cashing）技术

尽管我们都已经知道 HotSpot 的执行引擎采用的并非是基于寄存器的架构，但这并不代表 HotSpot VM 的实现并没有间接利用到寄存器资源。寄存器是物理 CPU 中的组成部分之一，它同时也是 CPU 中非常重要的高速存储资源。一般来说，寄存器的读/写速度非常迅速，甚至可以比内存的读/写速度快上几十倍不止，不过寄存器资源却非常有限，不同平台下的 CPU 寄存器数量是不同和不规律的。寄存器主要用于缓存本地机器指令、数值和下一条需要被执行的指令地址等数据。

笔者在 8.2.2 节中曾详细分析过寄存器架构和栈式架构之间的区别，尽管基于栈式架构的虚拟机所使用的零地址指令更加紧凑，但完成一项操作的时候必然需要使用更多的入栈和出栈指令，这同时也就意味着将需要更多的指令分派（instruction dispatch）次数和内存读/写次数。由于操作数是存储在内存中的，因此频繁地执行内存读/写操作必然会影响执行速度。为了解决这个问题，JVM 的设计者们提出了栈顶缓存（ToS，Top-of-Stack Cashing）技术，将栈顶元素全部缓存在物理 CPU 的寄存器中，以此降低对内存的读/写次数，提升执行引擎的执行效率。

在 HotSpot VM 中，TosState 用于描述指令执行前后 Top-of-Stack 的状态，Top-of-Stack 可以被缓存在一个或多个寄存器中。缓存值 TosState 对应数据的机器级描述。HotSpot VM 一共包含 9 种 TosState，如下所示：

代码 8-6 TosState

```
// TosState describes the top-of-stack state before and after the
// execution of a bytecode or method. The top-of-stack value may be cached
// in one or more CPU registers. The TosState corresponds to the 'machine
// represention' of this cached value. There's 4 states corresponding
// to the JAVA types int, long, float & double as well as a 5th state
// in case the top-of-stack value is actually on the top of stack
// (in memory) and thus not cached. The atos state corresponds to the
// itos state when it comes to machine representation but is used
// separately for (oop) type specific operations
// (e.g. verification code).
enum TosState {         // describes the tos cache contents
    btos = 0,            // 栈顶缓存 byte/bool 类型数据
    ctos = 1,            // 栈顶缓存 char 类型数据
    stos = 2,            // 栈顶缓存 short 类型数据
    itos = 3,            // 栈顶缓存 int 类型数据
    ltos = 4,            // 栈顶缓存 long 类型数据
    ftos = 5,            // 栈顶缓存 float 类型数据
    dtos = 6,            // 栈顶缓存 double 类型数据
    atos = 7,            // 栈顶缓存 object 类型数据
    vtos = 8,            // 栈顶缓存 tos 类型数据
    number_of_states,
    ilgl                // illegal state: should not occur
};
```

从虚拟机的概念模型上来说，当执行一项 int 类型的加法运算时，操作数栈中必然会伴随着频繁的入栈和出栈操作。其中“iadd”指令用于将操作数栈中的 2 个 int 类型的栈顶元素出栈，执行运算后再将 int 类型的运算结果重新压入栈顶。在此大家需要注意，在操作数栈中的数据必须进行正确的操作，比如不能够在入栈 2 个 int 类型的数值后，却把它们当做 long 类型的数值去操作，或者入栈 2 个 double 类型的数值后，使用“iadd”指令对它们执行加法运算等情况出现。之前也说过，栈顶缓存技术就是用于降低对内存的读/写次数，提升执行引擎的执行效率，那么在 HotSpot VM 的源码实现中，栈顶元素究竟是如何执行存/取操作的呢？本书示例了包含在/openjdk/hotspot/src/cpu/x86/vm/templateTable_x86_32.cpp 中的 iop2 指令模板，如下所示：

代码 8-7 iop2 指令模板

```
void TemplateTable::iop2(Operation op) {
  transition(itos, itos);
  switch (op) {
    case add  :                    __ pop_i(rdx); __ addl (rax, rdx); break;
    case sub  : __ mov(rdx, rax); __ pop_i(rax); __ subl (rax, rdx); break;
    case mul  :                    __ pop_i(rdx); __ imull(rax, rdx); break;
```

```
      case _and :                  __ pop_i(rdx); __ andl (rax, rdx); break;
      case _or  :                  __ pop_i(rdx); __ orl  (rax, rdx); break;
      case _xor :                  __ pop_i(rdx); __ xorl (rax, rdx); break;
      case shl  : __ mov(rcx, rax); __ pop_i(rax); __ shll (rax);      break;
      case shr  : __ mov(rcx, rax); __ pop_i(rax); __ sarl (rax);      break;
      case ushr : __ mov(rcx, rax); __ pop_i(rax); __ shrl (rax);      break;
      default   : ShouldNotReachHere();
    }
  }
```

在上述代码示例中，transition()函数的作用就是校验执行前后的 TosState 是否符合要求。也就是说，在执行“iadd”指令之前，TosState 必须确保是 itos，而执行运算之后的运算结果也同样必须是 itos。当通过校验后，如果匹配是加法运算指令时，便会将 rdx 寄存器和 rax 寄存器中的缓存值相加，最后将运算结果缓存在 rax 寄存器中后返回。由于栈顶元素并非缓存在内存中，因此也就无需对内存频繁地执行读/写操作了。

8.2.6 实战：跟踪字节码解释器的执行步骤

在 8.1.2 节中，笔者曾详细地描述过执行引擎是如何根据字节码指令执行的，当然这仅仅只是一种概念模型，不同 JVM 的实现细节不尽相同。比如在 HotSpot 虚拟机中，尽管操作数栈式作为执行引擎的一个运行时工作区，临时数据虽然缓存在操作数栈中，但为了性能考虑，栈顶元素均全部缓存在物理 CPU 的寄存器中。因此概念模型所描述的执行引擎的执行步骤与实际的执行步骤之间多少是存在一定的差别，当然有一点是可以肯定的，就是无论采用任何优化技术，从本质上来说，HotSpot 虚拟机自始至终都是基于栈式架构的。所以为了让大家更好地理解执行引擎的执行步骤，笔者将会通过本章的实战小节把代码 8-2 以图文并茂的形式为大家叙述基于概念模型的执行引擎的执行步骤。如图 8-4 至图 8-9 所示。

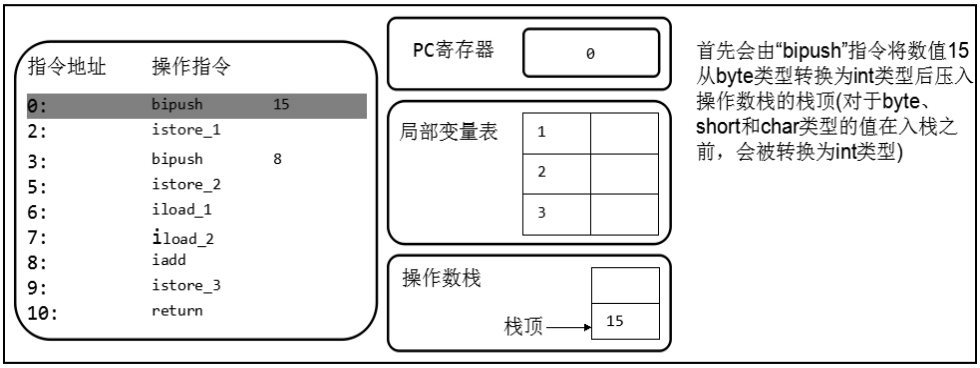


图 8-4 执行指令地址为 0 的操作指令

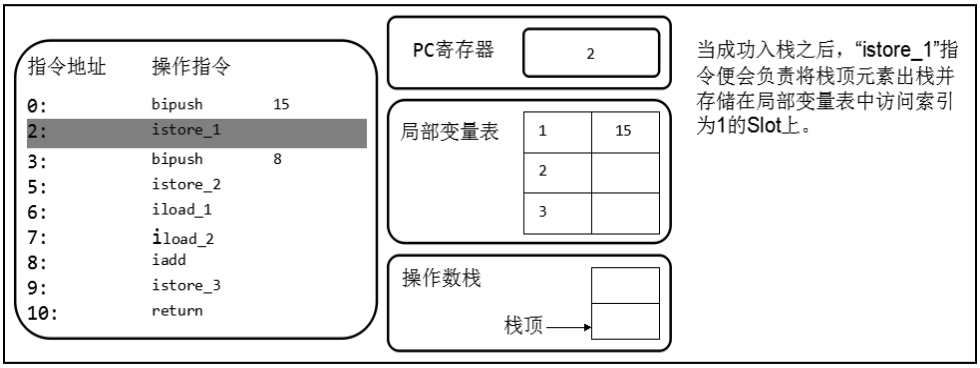


图 8-5 执行指令地址为 2 的操作指令

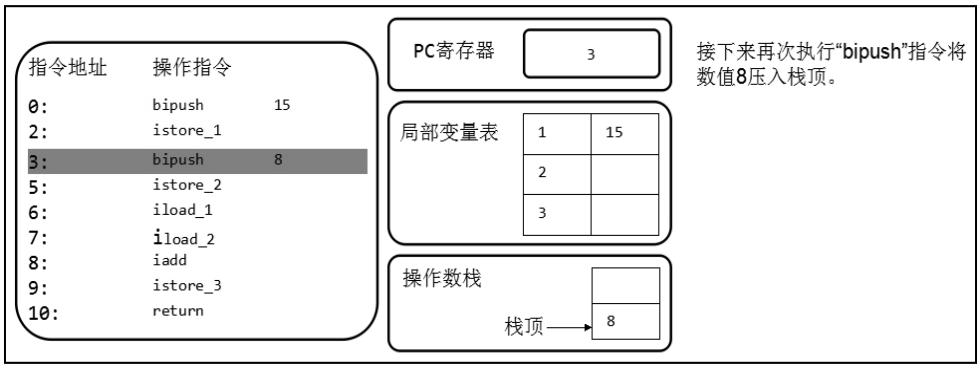


图 8-6 执行指令地址为 3 的操作指令

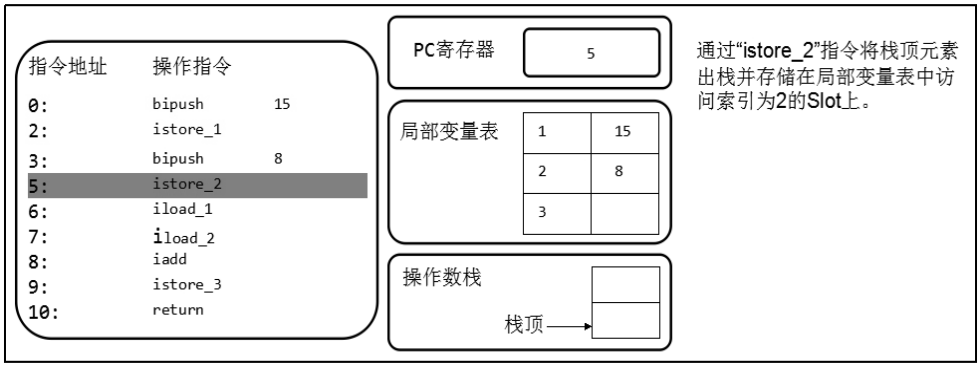


图 8-7 执行指令地址为 5 的操作指令

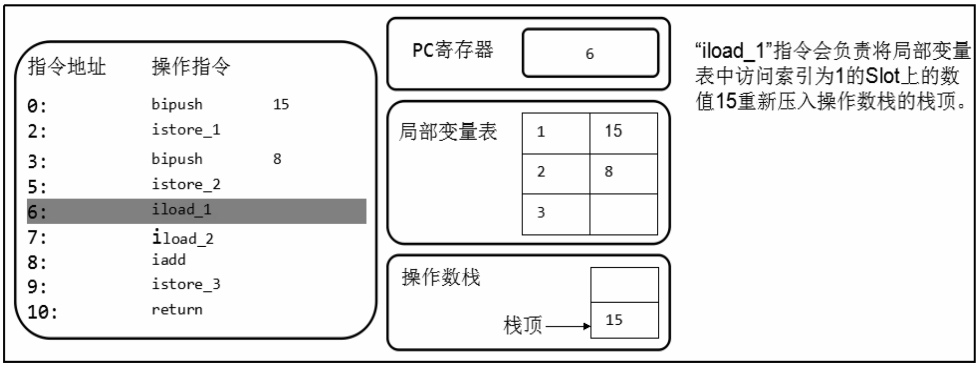


图 8-8 执行指令地址为 6 的操作指令

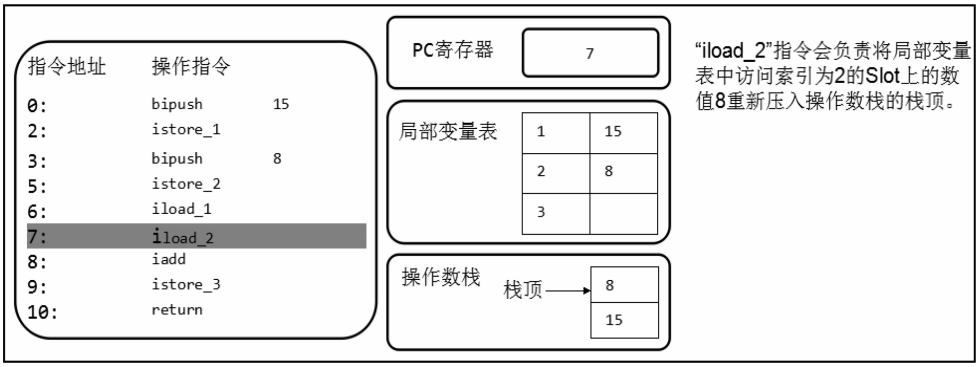


图 8-9 执行指令地址为 7 的操作指令

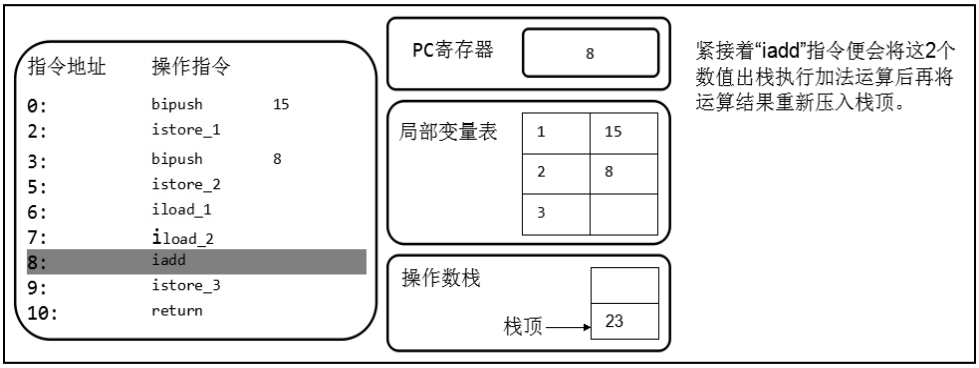


图 8-10 执行指令地址为 8 的操作指令

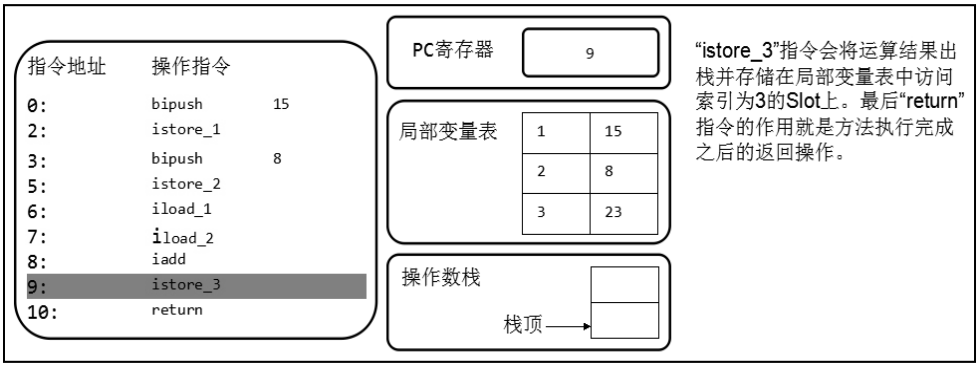


图 8-11 执行指令地址为 9 的操作指令

8.3 解释器与JIT编译器

早在 Java1.0 版本的时候，Sun 公司发布了一款名为 Sun Classic VM 的 Java 虚拟机，它同时也是世界上第一款商用 Java 虚拟机，在当时这款虚拟机内部只提供解释器，用今天的眼光来看待必然是效率低下的，因为如果 Java 虚拟机只能够在运行时对代码采用逐行解释执行，程序的运行性能可想而知。但是如今的 HotSpot VM 中不仅内置有解释器，还内置有先进的 JIT（Just In Time Compiler）编译器，在 Java 虚拟机运行时，解释器和即时编译器能够相互协作，各自取长补短。在此大家需要注意，无论是采用解释器进行解释执行，还是采用即时编译器进行编译执行，最终字节码都需要被转换为对应平台的本地机器指令。或许有些开发人员会感觉到诧异，既然 HotSpot VM 中已经内置 JIT 编译器了，那么为什么还需要再使用解释器来“拖累”程序的执行性能呢？比如 JRockit VM 内部就不包含解释器，字节码全部都依靠即时编译器编译后执行，尽管程序的执行性能会非常高效，但程序在启动时必然需要花费更长的时间来进行编译。对于服务端应用来说，启动时间并非是关注重点，但对于那些看中启动时间的应用场景而言，或许就需要采用解释器与即时编译器并存的架构来换取一个平衡点。

既然 HotSpot VM 中采用了即时编译器，那么这就意味着将字节码编译为本地机器指令是一件运行时任务。在 HotSpot VM 中内嵌有两个 JIT 编译器，分别为 Client Compiler 和 Server Compiler，但大多数情况下我们简称为 C1 编译器和 C2 编译器。开发人员可以通过如下命令显式指定 Java 虚拟机在运行时到底使用哪一种即时编译器，如下所示：

- -client：指定 Java 虚拟机运行在 Client 模式下，并使用 C1 编译器；
- -server：指定 Java 虚拟机运行在 Server 模式下，并使用 C2 编译器。

除了可以显式指定 Java 虚拟机在运行时到底使用哪一种即时编译器外，默认情况下

HotSpot VM 则会根据操作系统版本与物理机器的硬件性能自动选择运行在哪一种模式下，以及采用哪一种即时编译器。简单来说，C1 编译器会对字节码进行简单和可靠的优化，以达到更快的编译速度；而 C2 编译器会启动一些编译耗时更长的优化，以获取更好的编译质量。不过在 Java7 版本之后，一旦开发人员在程序中显式指定命令“-server”时，缺省将会开启分层编译（Tiered Compilation）策略，由 C1 编译器和 C2 编译器相互协作共同来执行编译任务。不过在早期版本中，开发人员则只能够通过命令“-XX:+TieredCompilation”手动开启分层编译策略。

之前笔者曾经提及过，缺省情况下 HotSpot VM 是采用解释器与即时编译器并存的架构，当然开发人员可以根据具体的应用场景，通过命令显式地为 Java 虚拟机指定在运行时到底是完全采用解释器执行，还是完全采用即时编译器执行。如下所示：

- -Xint：完全采用解释器模式执行程序；
- -Xcomp：完全采用即时编译器模式执行程序；
- -Xmixed：采用解释器+即时编译器的混合模式共同执行程序。

在此大家需要注意，如果 Java 虚拟机在运行时完全采用解释器执行，那么即时编译器将会停止所有的工作，字节码将完全依靠解释器逐行解释执行。反之如果 Java 虚拟机在运行时完全采用即时编译器执行，但解释器仍然会在即时编译器无法进行的特殊情况下介入执行，以确保程序能够最终顺利执行。

由于即时编译器将本地机器指令的编译推迟到了运行时，自此 Java 程序的运行性能已经达到了可以和 C/C++程序一较高下的地步。这主要是因为 JIT 编译器可以针对那些频繁被调用的“热点代码”做出深度优化，而静态编译器的代码优化则无法完全推断出运行时热点，因此通过 JIT 编译器编译的本地机器指令比直接生成的本地机器指令拥有更高的执行效率也就理所当然了。比如使用 Python 实现的 PyPy 执行器，比使用 C 实现的 CPython 解释器更加灵活，更重要的是，在程序的运行性能上进行比较，PyPy 将近是 CPython 解释器执行效率的 1 至 5 倍，这就是对 JIT 技术魅力的一个有力证明。并且 Java 技术自身的诸多优势同样也是 C/C++无法比拟的，所谓各有所长就是这个道理。在此大家需要注意，世界上永远没有最好的编程语言，只有最适用于具体应用场景的编程语言。

8.3.1 查阅HotSpot的运行时执行模式

尽管从 Java5 版本开始，缺省情况下 HotSpot VM 会根据操作系统版本与物理机器的硬件性能自动选择究竟是运行在 Client 模式下，还是运行在 Server 模式下，不过物理机器的性能究竟要达到一个什么样的标准才算是符合“服务器级别”的运行环境标准呢？简单来说，如果走双核或者以上，并且内存在 2GB 或者以上的物理机器，HotSpot VM 就会认定符合“服

务器级别”的标准，因此缺省也就会以 Server 模式运行。在此大家需要注意，如果是 32 位的 Windows 平台，HotSpot VM 在任何情况下都以 Client 模式运行。

如果大家希望查阅当前 HotSpot VM 在运行时缺省是以什么样的模式运行，可以通过命令“java -version”进行查阅，如下所示：

代码 8-8 执行命令“java -version”

```
java version "1.7.0_15"
Java(TM) SE Runtime Environment (build 1.7.0_15-b03)
Java HotSpot(TM) Client VM (build 23.7-b01, mixed mode, sharing)
```

由于笔者的物理机器上所安装的操作系统为 32 位的 Windows，因此 HotSpot VM 将会选择以 Client 模式运行。当然如果是希望以 Server 模式运行，则可以显式地通过命令“java -server”强制 HotSpot VM 以 Server 模式运行。如下所示：

代码 8-9 执行命令“java -server -version”

```
java version "1.7.0_15"
Java(TM) SE Runtime Environment (build 1.7.0_15-b03)
Java HotSpot(TM) Server VM (build 23.7-b01, mixed mode, sharing)
```

如代码 8-8 和代码 8-9 所示，HotSpot VM 在默认情况下都是采用解释器+即时编译器的混合模式共同执行程序。在 8.3 节中，开发人员可以通过命令显式指定 Java 虚拟机究竟是采用解释器还是即时编译器的方式执行程序。如下所示：

代码 8-10 执行命令“-Xint”和“-Xcomp”

```
>java -Xint -version
java version "1.7.0_15"
Java(TM) SE Runtime Environment (build 1.7.0_15-b03)
Java HotSpot(TM) Client VM (build 23.7-b01, interpreted mode, sharing)

>java -Xcomp -version
java version "1.7.0_15"
Java(TM) SE Runtime Environment (build 1.7.0_15-b03)
Java HotSpot(TM) Client VM (build 23.7-b01, compiled mode, sharing)
```

如果开发人员并没有显式指定有命令“-Xint”或“-Xcomp”的话，HotSpot VM 默认采用“-Xmixed”混合模式。

8.3.2 解释器的工作机制与构成模块

在 1.2.2 节中，笔者曾经分析过 Java 程序实现跨平台的基石就是字节码。在那个年代，

JVM 设计者们的初衷仅仅只是单纯地为了满足 Java 程序实现跨平台特性，因此避免采用静态编译的方式直接生成本地机器指令，从而诞生了实现解释器在运行时采用逐行解释字节码执行程序的想法。由于解释器在设计和实现上非常简单，因此除了 Java 语言之外，还有许多高级语言同样也是基于解释器执行的，比如 Python、Perl、Ruby 等。但是在今天，基于解释器执行已经沦落为低效的代名词，并且时常被一些 C/C++程序员所调侃，即便现在 HotSpot VM 中已经内嵌先进的 JIT 编译器，但 Java 程序的执行性能或多或少地还是被一些并不真正了解 Java 的开发人员所误解，不过无论如何，基于解释器的执行模式仍然为中间语言的发展做出了不可磨灭的贡献。

对于大多数 Java 开发人员而言，对解释器的执行机制或许并不会感觉到陌生。简单来说，当 Java 虚拟机启动时会根据预定义的规范对字节码采用逐行解释的方式执行，解释器真正意义上所承担的角色就是一个运行时“翻译者”，将字节码文件中的内容“翻译”为对应平台的本地机器指令执行。当一条字节码指令被解释执行完成后，接着再根据 PC 寄存器中记录的下一条需要被执行的字节码指令执行解释操作，这就是解释器的工作机制，同样也是解释器的工作任务。

解释器的任务就是负责将字节码指令解释为对应平台的本地机器指令执行。在 HotSpot VM 中，解释器主要由 Interpreter 模块和 Code 模块构成。其中 Interpreter 模块实现了解释器的核心功能，该模块中主要包括了两种类型的解释器，分别为模板解释器和 C++解释器。而 Code 模块主要用于管理 HotSpot VM 在运行时生成的本地机器指令。那么本书接下来将会围绕这两个模块展开讲解。

Interpreter 模块

Interpreter 模块实现了解释器的核心功能，该模块中主要包括了两种类型的解释器，分别为模板解释器和 C++解释器。其中模板解释器官方版本正在使用，而 C++解释器官方版本却并没有在使用。在 HotSpot VM 中，模板解释器和 C++解释器分别由 Interpreter 模块中的 TemplateInterpreter 子模块和 CppInterpreter 子模块实现。除此之外，Interpreter 模块中还包含一个非常重要的子模块，那就是 AbstractInterpreter 模块，该模块中定义了基于汇编模型的解释器和解释器生成器的抽象行为，以及还包含了一些与平台无关的公共操作。

Code 模块

Code 模块主要用于管理 HotSpot VM 在运行时生成的本地机器指令。在 Code 模块中，不得不提及的就是 CodeCache 子模块，该模块也被称之为代码缓存模块，主要用于缓存由 HotSpot VM 在运行时生成的本地机器指令。当 Java 虚拟机启动时，会在内存空间中为其分配一块内存区域，专门用于缓存本地机器指令，该空间与方法区一起被合称为非堆内存。当然开发人员可以通过如下选项设置代码缓存区的内存大小，如下所示：

表 8-1 代码缓存区的内存选项配置

选项	缺省值	描述	备注
-XX:CodeCacheExpansionSize	与平台相关	设置代码缓存区扩展大小的参数	暂无
-XX:InitialCodeCacheSize	与平台相关	设置代码缓存区的初始内存	暂无
-XX:ReservedCodeCacheSize	与平台相关	设置代码缓存区的最大内存	暂无

8.3.3 JIT编译器的工作机制与构成模块

HotSpot VM的执行引擎主要由解释器与JIT编译器构成，并且在运行时如果开发人员并没有通过选项“-Xint”或“-Xcomp”显式指定执行模式，那么HotSpot VM缺省将会以采用解释器+即时编译器的混合模式共同执行程序。当然是否需要启动JIT编译器将字节码直接编译为对应平台的本地机器指令，则需要根据代码被调用执行的频率而定。关于那些需要被编译为本地代码的字节码，也被称之为“热点代码”，JIT编译器在运行时会针对那些频繁被调用的“热点代码”做出深度优化，将其直接编译为对应平台的本地机器指令，以此提升Java程序的执行性能，这就是为什么HotSpot VM会被称之为HotSpot的原因。解释器与JIT编译器混合执行流程，如图 8-12 所示[②] ：

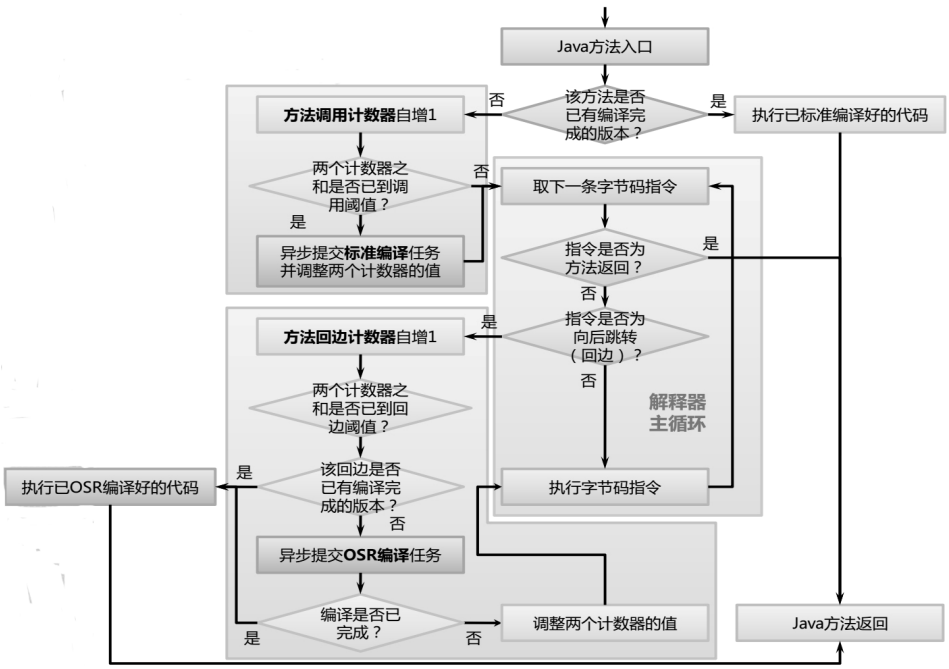


图 8-12 Client VM 模式下的解释执行与编译执行

② 图片来源于 RednaxelaFX 的《JVM 分享：Java Program in Action》PPT。

那么“热点代码”的衡量标准究竟是什么呢？也就是说，一段代码究竟需要被执行多少次，才能被标记为“热点代码”然后经由 JIT 编译器执行编译呢？关于“热点代码”的具体探测方式，请阅读 8.3.5 节。

在 HotSpot VM 中，JIT 编译器主要由 C1 模块、Opto 模块和 Shark 模块构成。其中 C1 模块实现了 Client Compiler 编译器；而 Opto 模块实现了 Server Compiler 编译器，因此 C2 编译器也可以被称之为 Opto 编译器；Shark 模块中则实现了一个基于 LLVM 的编译器。在此大家需要注意，本章所有关于 JIT 编译器的内容都仅针对 C1 编译器，而关于 C2 编译器与 Shark 模块中的内容，大家则可以参考其他 JVM 文档或者直接阅读 HotSpot VM 的源码。

8.3.4 分层编译策略

之前笔者曾经提及过，缺省情况下 HotSpot VM 会根据操作系统版本与物理机器的硬件性能自动选择运行在哪一种模式下，以及采用哪一种即时编译器。如果开发人员通过命令“-client”显式指定 Java 虚拟机运行在 Client 模式下，那么就将会使用 C1 编译器，反之则运行在 Server 模式下，并使用 C2 编译器。不过在 Java7 版本之后，一旦开发人员在程序中显式指定有命令“-server”时，缺省将会开启分层编译（Tiered Compilation）策略，由 C1 编译器和 C2 编译器相互协作共同来执行编译任务。分层编译策略将会根据编译器执行编译、优化的规模与耗时，划分出不同的编译层次，如下所示：

- 第 0 层：程序由解释器解释执行，但解释器并不开启性能监控功能，可触发第一层编译；
- 第 1 层：程序由 C1 编译器编译为本地机器指令执行，C1 编译器会对字节码进行简单和可靠的优化，以达到更快的编译速度；
- 第 2 层：程序由 C2 编译器编译为本地机器指令执行，但 C2 编译器会启动一些编译耗时更长的优化（代码将有可能被重复编译多次），甚至有可能根据性能监控信息进行一些不可靠的激进优化；
- 第 3 层：程序由 C1 编译器编译为本地机器指令执行，采集性能数据进行优化措施；
- 第 4 层：程序由 C2 编译器编译为本地机器指令执行，进行完全优化。

包含在 hotspot/src/share/vm/utilities/globalDefinitions.hpp 中的枚举类型 CompLevel 定义了 HotSpot VM 的编译层次，如下所示：

代码 8-11 globalDefinitions.hpp 中定义的编译层次

```
enum CompLevel {
  CompLevel_any             = -1,
  CompLevel_all              = -1,
  CompLevel_none             = 0,        // Interpreter
```

```
    CompLevel_simple            = 1,    // C1
    CompLevel_limited_profile   = 2,   // C1, invocation & backedge counters
    CompLevel_full_profile      = 3,   // C1, invocation & backedge counters
+ mdo
    CompLevel_full_optimization = 4,  // C2 or Shark
    ... ...
  };
```

当 Java 虚拟器启动时，解释器可以首先发挥作用，而不必等待即时编译器全部编译完成后再执行，这样可以省去许多不必要的编译时间，即工作在第 0 层。接下来将会由编译策略模块决定到底是启用第 2 层还是第 3 层编译，当在第 3 层完成性能采集后，将会逐步过渡到第 4 层编译。在此大家需要注意，编译过程中，并不是所有的“热点代码”都需要经由第 4 层执行编译，因为如果是一些并不重要的代码，将会直接在第 1 层执行编译。

8.3.5 热点探测功能

究竟什么是“热点代码”？简单来说，一个被多次调用的方法，或者是一个方法体内部循环次数较多的循环体都可以被称之为“热点代码”，因此都可以通过 JIT 编译器编译为本地机器指令。对于那些被频繁调用的方法，JIT 编译器会将整个方法都作为编译目标并将其编译为本地机器指令。但是对于方法体内部循环次数较多的循环体，JIT 编译器的编译目标同样也是针对整个方法而非循环体，由于这种编译方式发生在方法的执行过程中，因此也被称之为栈上替换，或简称为 OSR（On Stack Replacement）编译。

尽管频繁可以作为衡量“热点代码”的标准，但是一个方法究竟要被调用多少次，或者一个循环体究竟需要执行多少次循环才可以达到这个标准？也就是说，必然需要一个明确的阈值，JIT 编译器才会将这些“热点代码”编译为本地机器指令执行。衡量一段代码是否是“热点代码”主要依靠热点探测功能，目前 HotSpot VM 所采用的热点探测方式是基于计数器的热点探测。

采用基于计数器的热点探测，HotSpot VM 将会为每一个方法都建立 2 个不同类型的计数器，分别为方法调用计数器（Invocation Counter）和回边计数器（Back Edge Counter）。其中方法调用计数器用于统计方法的调用次数，而回边计数器则用于统计循环体执行的循环次数。一旦某个方法或者循环体的执行次数超出缺省所指定的阈值时，就可以被认定为“热点代码”，因此就可以被 JIT 编译器执行编译操作。基于计数器的热点探测在实现上稍显麻烦，因为需要为每一个方法都建立和维护两个私有的计数器，而且还不能直接获取方法的调用关系，不过采用这种方式相对来说却更加精准，大家权衡好利与弊，只不过是一种非常明显判定“热点代码”的探测方式。

笔者先从方法调用计数器开始讲起。在缺省情况下，Client VM 模式下的方法调用次数

超过 1500 次时，将会触发 JIT 编译；而在 Server VM 模式下，方法的调用次数超过 10000 次时，才会触发 JIT 编译。当然开发人员可以通过选项“-XX:CompileThreshold”来设置方法调用计数器的阈值。如图 8-12 所示，当一个方法被调用时，第一步就是检查当前方法是否已经被 JIT 编译器执行过编译，是否存在编译后的版本，如果存在的话，则执行已经编译好的本地机器指令。如果不存在 JIT 编译后的版本，那么则将方法调用计数器的值加 1，然后判断方法调用计数器的值与回边计数器的值相加之和是否超过方法调用计数器的阈值，如果超过的话，将会向 JIT 编译器提交一个对当前方法执行编译的请求。

当然方法调用计数器中所统计的方法调用次数并非是一个绝对次数，而是一个瞬时次数。也就是说，方法调用计数器的值仅仅只会在某一段时间之内有效，一旦超出所规定的时间阈值时，如果当前方法被调用的次数仍然达不到编译标准，那么这个方法计数器的值就会消退 50%，这个消退过程也就被称之为方法调用计数器的热度消退，而这段时间就被称之为方法调用次数统计的消退周期。进行热度消退的动作是在进行 GC 时顺带一起执行的，当然开发人员可以通过选项“-XX:-UseCounterDecay”来关闭缺省的热度消退，这样一来方法调用计数器所统计的就会是方法被调用的绝对次数。除了可以使用选项“-XX:-UseCounterDecay”来关闭热度消退外，还可以使用选项“-XX:CounterHalfLifeTime”设置消退周期的时间阈值。

在此大家需要注意，如果程序中并没有显式设置“-XX:-BackgroundCompilation”或“-Xbatch”选项时，HotSpot VM 的执行引擎缺省将会按照异步模式在后台执行编译任务，程序的工作线程将不会出现暂停，而是按照解释执行的方式继续执行程序，直到提交的编译请求全部都被 JIT 编译器编译完成后，再异步安装编译结果执行已经编译好的本地机器指令。反之如果程序中显式设置这 2 个选项时，则意味着将会采用同步模式执行编译任务，这样一来一旦计数器提交有编译任务时，程序的工作线程将会出现暂停，直到 JIT 编译器完成编译之后，程序才会恢复执行。如图 8-13 所示。

当大家明白方法调用计数器之后，接下来笔者再来为大家讲解另外一个计数器：回边计数器。在缺省情况下，Client VM 模式下的循环次数超过 993 次时，将会触发 JIT 编译；而在 Server VM 模式下，循环次数超过 10700 次时，才会触发 JIT 编译。尽管 HotSpot VM 为回边计数器提供了一个类似于设置方法调用计数器阈值的选项“-XX:BackEdgeThreshold”，但是当前 HotSpot VM 中实际上并未使用该选项，因此开发人员只能够通过另外一个选项“-XX:OnStackReplacePercentage”来间接地调整回边计数器的阈值，其计算公式如下：

- Client VM 模式下，回边计数器的阈值计算公式：

  (CompileThreshold * OnStackReplacePercentage) / 100

- Server VM 模式下，回边计数器的阈值计算公式：

  (CompileThreshold * (OnStackReplacePercentage - InterpreterProfilePercentage)) / 100;

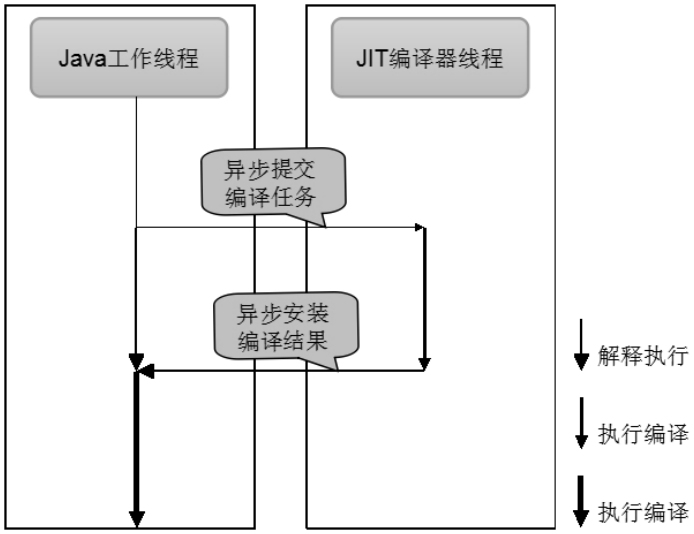


图 8-13(a) 异步提交编译任务与安装编译结果

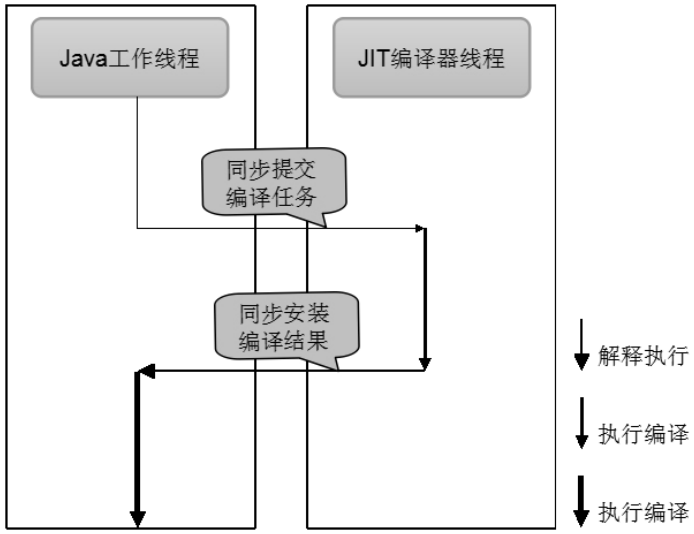


图 8-13(b) 同步提交编译任务与安装编译结果

为了使大家阅读更方便，本书示例了根据计数器触发编译条件的代码片段，如下所示：

代码 8-12 触发编译的条件

```
class InvocationCounter {
    private:                          // bit no: |31  3|  2  | 1 0 |
        unsigned int _counter;        // format: [count|carry|state]
        enum PrivateConstants {
```

```
                number_of_state_bits = 2,
                number_of_carry_bits = 1,
                number_of_noncount_bits = number_of_state_bits +
                number_of_carry_bits,
                number_of_count_bits = BitsPerInt -
                number_of_noncount_bits,
                // ...
            };
        // ...
    };
    void InvocationCounter::reinitialize(bool delay_overflow) {
        InterpreterInvocationLimit =CompileThreshold
          << number_of_noncount_bits;
        InterpreterProfileLimit =
            ((CompileThreshold * InterpreterProfilePercentage) / 100)
          << number_of_noncount_bits;
        // ...
        if(ProfileInterpreter) {
            InterpreterBackwardBranchLimit =
                 (CompileThreshold * (OnStackReplacePercentage-
                  InterpreterProfilePercentage)) / 100;
        } else {
            InterpreterBackwardBranchLimit =
                 ((CompileThreshold * OnStackReplacePercentage) / 100)
          << number_of_noncount_bits;
        }
    }
```

如图 8-12 所示，当解释器遇到一条回边指令时，首先会和方法调用计数器一样检查即将执行的代码片段是否存在编译后的版本，如果存在的话，则执行已经编译好的本地机器指令。如果不存在 JIT 编译后的版本，那么则将回边计数器的值加 1，然后判断方法调用计数器的值与回边计数器的值相加之和是否超过回边计数器的阈值，如果超过的话，将会提交一个 OSR 编译请求，并且把回边计数器的值降低一些，以便继续在解释器中执行循环，等待 JIT 编译器的编译结果。

在此大家需要注意，由于回边计数器并没有和方法调用计数器一样存在热度消退的过程，因此回边计数器所统计的就是当前方法体内部循环体执行循环的绝对次数。

8.4 本章小结

想要了解执行引擎的工作原理，必然首先需要对栈帧非常熟悉。本章笔者详细地讲解了局部变量表、操作数栈、动态链接以及方法返回值等栈帧的几个组成结构后，又为大家详细地比较了基于寄存器架构和栈式架构的虚拟机在设计和实现上的区别。考虑到频繁访问操作

数栈将会造成运行性能的下降，因此笔者又从源码的角度出发，为大家介绍了 HotSpot 中的栈顶缓存（ToS，Top-of-Stack Cashing）技术是如何将栈顶元素全部缓存在物理 CPU 的寄存器中。除了为大家介绍 HotSpot VM 的架构模型外，笔者还对解释器和 JIT 编译器的工作流程进行了系统的分析，并重点讲解了 JIT 编译器的分层编译策略和“热点代码”的探测方式。

本章是本书的结尾，当大家阅读完本书的所有内容后，仍然还需要结合 HotSpot VM 的源码进行阅读和分析，这样才能加深对底层原理的认识和理解。

附录 A

Java7新增语法特性

Java7 在语法层面上所带来的改变是非常大的，Java 的设计者们为开发人员在早期的语法特性上扩充了不少的新增语法特性，比如：try-with-resources 语句、泛型的“<>”类型推断运算符、声明二进制字面值、字面值下画线支持、switch 表达式支持 String 类型、mutil-catch 特性、NIO2.0 文件系统的改变等。本书的附录 A 将会针对这些新增语法特性进行讲解，让大家深刻理解 Java7 在语法层面上所带来的改变和在实际开发过程中使用这些新特性所带来的诸多好处。

A.1 try-with-resources语句

在实际的开发过程中，开发人员经常需要在程序中消费如 I/O 连接、DataBase 连接、Socket 连接等资源，并且在消费结束后，还需要进行资源的回收工作。如果这些资源没有被及时回收，就会一直占用 Java 虚拟机中的内部资源和外围系统的有限资源，从而极大程度地影响了资源分配操作。

在 Java7 之前，开发人员可以在程序中定义 finally 子句实现手动的资源回收，如下所示：

代码 A-1 手动资源回收

```
    try {
        /* 资源初始化并使用资源 */
    } finally {
        /* 资源回收工作 */
}
```

之所以选用 finally 子句执行资源的回收工作，最主要的原因是依赖于 finally 子句的“必须执行”机制。也就是说，哪怕程序在运行过程中发生异常，也不会导致资源在消费结束后得不到有效回收。虽然 finally 子句能够帮助开发人员完成资源回收，但对于刚接触 Java 的

开发人员而言，却时常忘记消费完资源后，定义 finally 子句，或者在非 finally 子句代码块内执行资源回收操作，这样一来手动资源回收模式却因为开发人员的“懒散”失去了自身优势。

那么是否存在一种方式能够确保资源在消费结束后，哪怕是忘记手动回收，也能够实现自动资源管理？就像 Java 虚拟机的自动内存管理机制一样，开发人员只需关注业务即可，从而无需再关心无用对象何时回收、如何回收。早在 Java5 的时候，所有可回收资源类型都缺省实现了 Closeable 接口，并重写了该接口内部的 close()方法，以此实现资源回收操作，Java7 对 Closeable 接口做了一些调整，如下所示：

代码 A-2 Closeable 接口

```
/**
 * A {@code Closeable} is a source or destination of data that can be closed.
 * The close method is invoked to release resources that the object is
 * holding (such as open files).
 *
 * @since 1.5
 */
public interface Closeable extends AutoCloseable {
    /**
     * Closes this stream and releases any system resources associated
     * with it. If the stream is already closed then invoking this
     * method has no effect.
     *
     * @throws IOException if an I/O error occurs
     */
    public void close() throws IOException;
}
```

上述代码示例中，Closeable 接口继承了 AutoCloseable 接口。考虑到手动进行资源回收的弊端，Java 的设计者们在 Java7 中对 Closeable 接口进行了扩展，帮助开发人员能够顺利过渡到资源自动化管理时代。

当 JavaAPI 中所有可回收的资源类型都直接或者间接地实现 AutoCloseable 接口后，Java 的设计者们则在 Java7 的语法层面上提供了 try-with-resources 语句，以此实现自动资源管理。出于安全方面的考虑，资源回收操作一般都包含在异常处理代码块内部，那么 try-with-resources 语句则被定义在 try 语句的表达式中。也就是说，如今 try 语句能够支持定义表达式，却具有一定的局限性，因为定义在 try 语句表达式内部的资源对象必须派生于 AutoCloseable 接口。如下所示：

代码 A-3　自动资源管理

```
/**
 * 使用 try-with-resources 语句实现自动资源管理
 *
 * @author JohnGao
 */
public class AutoResourceManager1 {
  public static void main(String[] args) {
      try (BufferedReader read = new BufferedReader(new FileReader(
              "resource path"));) {
          read.readLine();
      } catch (IOException e) {
          e.printStackTrace();
      }
  }
}
```

上述代码示例中，由于使用了 try-with-resources 语句实现自动资源管理，所以开发人员将不必在消费完 BufferedReader 资源后定义 finally 子句进行手动资源回收操作。除此之外，try-with-resources 语句的表达式内部还能够定义多个资源对象，并进行统一多资源整合的自动管理。对于开发人员而言，这一切都是隐式进行的，无需手动参与。如下所示：

代码 A-4　多资源整合的自动资源管理

```
/**
 * 使用 try-with-resources 语句实现多资源整合的自动资源管理
 *
 * @author JohnGao
 */
public class AutoResourceManager2 {
   public static void main(String[] args) {
      try (BufferedReader read = new BufferedReader(new FileReader(
              "resource path"));
              BufferedWriter write = new BufferedWriter(new FileWriter(
                      "save path"));) {
          write.write(read.readLine());
      } catch (IOException e) {
          e.printStackTrace();
      }

  }

 }
```

上述程序示例中，try-with-resources 语句内部定义了 BufferedReader 和 BufferedWriter

这两个可回收资源类型，彼此之间通过符号“;”分隔。当程序执行结束后，自动资源管理将会对其进行资源回收操作。

在此大家需要注意，使用 try-with-resources 语句能够将开发人员从繁琐的体力劳动中“解放”出来。尽管已经顺利过渡到自动化资源管理时代，但为了在程序中避免因自动引发的各种异常和错误，我们仍需了解其背后的原理和机制，因为只有这样才能够确保不会迷失在自动所隐藏的美好“陷阱”中。

首先资源回收不等于对象回收，因为资源回收仅仅只会释放资源对象所占用的 Java 虚拟机中的内部资源和外围系统的有限资源，但并不会回收内存区中的对象实例。如下所示：

代码 A-5 自动资源管理后对象的存货状态

```
/**
 * 自动资源管理后对象的存活状态
 *
 * @author JohnGao
 */
public class AutoResourceManager3 {
   public static void main(String[] args) {
      BufferedReader read1 = null;
      try (BufferedReader read2 = new BufferedReader(new FileReader(
              "resource path"));) {
          read1 = read2;
      } catch (IOException e) {
          e.printStackTrace();
      }
      System.out.println("read1 所引用的对象实例是否已经执行回收->" + read1
== null);
   }
}
```

程序输出如下：

```
read1 所引用的对象实例是否已经执行回收->false
```

上述代码示例中，在 try-with-resources 语句内部将 read2 的引用赋值给 read1，并且当消费完资源 read2 后，自动资源管理便会执行资源回收操作。尽管执行了资源回收，但 read1 仍然持有之前 read2 的引用，由此可见资源回收确实不等于对象回收。

尽管在程序中使用 try-with-resources 语句非常方便，但是否需要因此废弃 finally 子句？finally 语句在 Java7 之前的绝大多数情况下都是用于手动资源回收操作，但根据实际的业务需求，finally 子句还能够应用于无论发生任何情况都“必须执行”的特殊场景下。

A.2 泛型的“<>”类型推断运算符

早在 Java5 的时候，Java 语法层面曾扩充了许多的新晋语法特性，其中泛型显得尤为重要。泛型是一种全新的语法风格，它的出现曾导致整个 Java API 都发生了较大改变，比如 Java 集合框架就使用了泛型语法进行重构。

最初在泛型语法没有出现之前，如果想要在程序中定义适用于任何数据类型的类、接口或者方法，这几乎是很难做到的。因为 Java 是一门强类型的编程语言，所以源代码在编译前期就必须确定所使用的具体数据类型。当然这并不意味着泛型语法没有出现之前，我们无法在程序中定义这种具有“通用”性质的数据类型。Object 作为所有类型的超类，开发人员可以选择使用 Object 作为“通用”的数据类型来定义类、接口或者方法。但是程序中过度依赖 Object 进行操作各种数据类型其实是非常不安全的，因为在某些特殊情况下，强制的类型转换需要将 Object 显式转换为所要操作的具体数据类型，由于类型不匹配的转换错误并不会发生在编译前期，所以我们根本无法确保程序在执行过程中的安全性。然而泛型的出现则很好地解决了这种弊端，并且同时确保了程序的安全性，甚至从某种意义上来说，泛型语法还能够有效提升代码的复用性。

那么究竟什么是泛型呢？泛型的核心概念其实就是参数化类型（parameterized type），所谓参数化类型，指的就是一个类、接口或者方法在编译期并不需要确定其具体类型，而是在运行期由类型参数动态指定。也就是说，在程序中使用了类型参数的类、接口和方法都可以被称之为泛型类、泛型接口和泛型方法。

在程序中定义一个泛型类型的实例时，传递给其类型参数的数据类型必须保证只能是引用类型（reference type），而不是原始类型（primitive type），否则将无法正确通过编译。但这并不代表开发人员无法在泛型代码中使用这些原始类型，因为我们可以利用 Java5 提供的自动装箱/拆箱机制，使用原始类型的封装类传递给泛型的类型参数，这一切对于开发人员而言几乎都是透明的。如下所示：

代码 A-6 泛型示例

```
/**
 * 泛型示例
 *
 * @author JohnGao
 */
public class GenericTest {
    public static void main(String[] args) {
        List<Integer> value = new ArrayList<Integer>();
        for (int i = 0; i < 10; i++)
            value.add(i);
```

```
        }
    }
```

上述代码示例中，集合接口 List 的类型参数为 Integer 类型，这也就意味着这个 List 集合所能存储的数据类型只能是 Integer 类型，而不允许存储其他数据类型。在实际的开发过程中，我们不仅需要在声明泛型时就指定类型参数的数据类型，并且在实例化泛型时，仍然需要再次重复指定类型参数的数据类型。那么是否存在一种机制能够简化泛型的类型参数定义？在 Java7 中，开发人员可以使用类型推断运算符简化类型参数的重复定义。

当声明一个泛型类型的时候，既然已经指定了其类型参数的数据类型，那么在实例化泛型时，我们则可以使用 Java7 提供的“<>”类型推断运算符简化类型参数的重复定义。类型推断运算符不仅可以简化类型参数的重复定义，还能够避免因前后类型参数的数据类型定义不同而发生编译错误，从而确保类型参数的统一。如下所示：

代码 A-7 使用类型推断运算符简化类型参数的重复定义

```
class Type<T> {
  T obj;
  public T getObj() {
      return obj;
  }
  Type(T obj) {
      this.obj = obj;
  }
}

  /**
   * 使用类型推断运算符简化类型参数的重复定义
   *
   * @author JohnGao
   */
public class GenericTest2 {
  /**
   * @author JohnGao
   */
  public static void main(String[] args) {
      /* 使用 Java7 之前的泛型语法定义类型参数的数据类型 */
      Type<String> type1 = new Type<String>("str");
      System.out.println("泛型类 Type 的类型参数为->"
              + type1.getObj().getClass().getName());

      /* 使用类型推断运算符简化类型参数的定义 */
      Type<Integer> type2 = new Type<>(1000);
      System.out.println("泛型类 Type 的类型参数为->"
              + type2.getObj().getClass().getName());
```

```
    }
}
```

程序输出如下：

```
泛型类 Type 的类型参数为->java.lang.String
泛型类 Type 的类型参数为->java.lang.Integer
```

上述代码示例中，在声明泛型类时指定了类型参数的数据类型，类型推断运算符便会在程序运行时自动推断实例化泛型类时应该匹配什么样的数据类型用于替换类型参数，这样的隐式替换操作，确实是为开发人员提供了便捷，有效地防止了错误的发生。

A.3 声明二进制字面值

进制数是人们利用符号进行计数的科学方法，对于计算机而言任何信息必须转换成二进制数，才能在计算机中进行存储和传输。进制数有很多种，然而在计算机中比较常见的有二进制、八进制、十进制和十六进制数。

Java 与 C/C++语言息息相关，因为 Java 语言继承了 C 语言的语法结构，并改编了 C++的对象模型。所以开发人员能够在程序中以最简单和最直接的方式去定义十进制、八进制和十六进制等字面值。随着 Java7 的正式来临，开发人员完全可以在程序中直接定义二进制字面值。当然程序在运行时的运算过程中，仍然会将非十进制的进制数转换成十进制进行输出。在此大家需要注意，二进制数是计算机中唯一能够被识别的数据，二进制数由 0 和 1 两个数字组成，运算规则是逢二进一。那么为了有效和其他进制数进行区分，二进制数用下标 2 或者数值末尾加字符 B 标识。比如二进制数 10110011 可以写成$(1011\ 0011)_2$或者 10110011B。在 Java 中定义一个二进制数时，字面值前缀必须为“0b”。如下所示：

代码 A-8 声明二进制字面值

```
int value1 = 0b10111001;
int value2 = 0b11111011;
```

Java 的语法规则除了允许开发人员直接在 Java 代码中声明进制数字面值外，Integer 类型还提供了十进制数转向其他进制数的方法。如下所示：

代码 A-9 不同进制数之间的转换

```
/**
 * 不同进制数之间的转换操作
 *
 * @author JohnGao
```

```
 */
public class NumberConvert {
    /**
     * @author JohnGao
     */
    public static void main(String[] args) {
        /* 定义十进制字面值 */
        int decimal = 0;
        /* 定义八进制数常量字面值 */
        final int OCTAL = 0123;
        /* 定义十六进制常量字面值 */
        final int HEX = 0x123AF;
        /* 定义二进制常量字面值 */
        final int BINARY = 0b010001101;
        decimal = OCTAL;
        System.out.println("Octal   -> Decimal: " + OCTAL);
        System.out.println("Decimal -> Octal: " + Integer.toOctalString
                           (decimal));
        decimal = HEX;
        System.out.println("Hex     -> Decimal: " + HEX);
        System.out.println("Decimal -> Hex: " + Integer.toHexString
                           (decimal));
        decimal = BINARY;
        System.out.println("Binary  -> Decimal: " + BINARY);
        System.out.println("Decimal -> Binary: " + Integer.
                           toBinaryString (decimal));
    }
}
```

程序输出如下：

```
Octal   -> Decimal: 83
Decimal -> Octal: 123
Hex     -> Decimal: 74671
Decimal -> Hex: 123af
Binary  -> Decimal: 141
Decimal -> Binary: 10001101
```

A.4 字面值下画线支持

Java7 在语法层面上的改变就连字面值的定义也考虑得非常细致。在上一个小节中本书详细地讲解了不同类型进制数之间的转换，其中涵盖了 Java7 新晋扩充的二进制字面值声明语法。由于进制数的呈现形式是整数类型，所以可以将其归纳到数值字面值这个大类中。

在程序中定义数值字面值是非常简单的，数值字面值由整数字面值和浮点数（浮点数由

单精度浮点数 float 和双精度浮点数 double 构成）字面值构成。如果根据业务需求，在程序中定义一个长度为 5 位的 int 类型字面值，或许在某些情况下，我们能够通过肉眼简单识别。但当定义一个长度比 int 类型更长的 long 类型字面值时，是否单凭肉眼就能够清晰分辨出所定义字面值的具体长度？考虑到数值字面值的可读性，Java7 允许使用符号“_”进行数值切分，这样一来，代码不仅可以有效提升可读性，还能够增强程序的后期维护性。如下所示：

代码 A-10　数值字面值切分

```
/* 切分整数字面值 */
long integer = 1000_0000_000L;
/* 切分单精度字面值 */
float singlePrecision = 1000_1000.1001F;
/* 切分双精度字面值 */
double doublePrecision = 1000_1000_1000.1001_1002D;
```

如果在程序中使用数值切分，那么 Java 前端编译器（JavaC 编译器）在将源代码编译成字节码时，会擦除 Java 代码中所有包含用于执行数值切分的符号“_”。也就是说，数值切分符号“_”并不会生成在字节码文件中，因为这样既无需更改 Java 虚拟机的内部指令集，又可以通过擦除特性将数值切分操作停留在编译前期。使用“javap -c”命令查看代码 A-10 中数值字面值切分示例编译后的字节码文件中的内容，如下所示：

代码 A-11　代码 A-10 编译后的字节码

```
Compiled from "Test.java"
public class Test {
  long integer;
  float singlePrecision;
  double doublePrecision;
  public Test();
    Code:
       0: aload_0
       1: invokespecial #1        // Method java/lang/Object."<init>":()V
       4: aload_0
       5: ldc2_w        #2        // long 100000000001
       8: putfield      #4        // Field integer:J
       11: aload_0
       12: ldc          #5        // float 1.0001E7f
       14: putfield      #6       // Field singlePrecision:F
       17: aload_0
       18: ldc2_w        #7       // double 1.0001000100010011E11d
       21: putfield      #9       // Field doublePrecision:D
       24: return
}
```

A.5 switch表达式支持String类型

Java 目前为开发人员提供了两种用于控制程序执行流程的多路分支语句：一种是开发人员非常熟悉的 if-else-if 语句；另外一种则是 switch 语句。虽然在实际的开发过程中开发人员使用 if-else-if 语句可以有效执行多路分支，然而在单条件的情况下使用 switch 语句执行更为高效，同时结构也会更清晰。

早在 Java7 之前，switch 语句的表达式值只能使用 byte、short、int 和 char 4 种原始数据类型，并且不允许定义重复的表达式值。但是随着 Java7 的正式来临，switch 语句则允许定义另一种全新的表达式值，那就是 String 类型。switch 语句结构清晰，同时使用方式也非常简单，因此在实际的开发过程中，如果是单条件的情况下，开发人员应该尽量使用 switch 多路分支语句去控制程序的执行流程。如下所示：

代码 A-12 switch 语句形式

```
switch (expression) {
case constant1:
      statement sequence
      break;
case constant2:
      statement sequence
      break;
case constant3:
      statement sequence
      break;
case constant4:
      statement sequence
      break;
default:
      statement sequence
}
```

上述代码示例中，会按照顺序结构从上到下依次检查是否有与表达式值相匹配的 case 值，如果程序执行到末尾都没有与表达式值相匹配的 case 值，则会执行 default 子句序列中的内容。在 switch 语句中，default 子句是可选的，如果没有该子句，并且表达式值与所有的 case 值都匹配失败时，就不会执行 switch 代码块中的任何内容。反之当匹配到一个 case 值的时候，就会执行该 case 值中的相关内容，直到程序执行完成并成功 break。这里有一点需要提醒大家，如果 case 块中并没有包含 break 时，程序则会依次执行到程序的末尾。

在 Java7 之前，开发人员如果想要在 switch 语句中定义字符类型的表达式值时，唯一的选择无疑是 char 类型。但是如果想直接使用字符串时，switch 语句就显得无能为力，因此

这无疑是之前的一种遗憾。但随着 Java7 的正式来临，开发人员如今可以在 switch 语句中使用字符串类型的表达式值，也就是说 switch 语句中可以直接使用 String 类型定义表达式值。如下所示：

代码 A-13　使用 String 类型定义 switch 语句的表达式值

```
    /**
     * 使用 String 类型定义 switch 语句的表达式值
     *
     * @author JohnGao
     */
public class SwitchTest {
    public static void main(String[] args) {
        /* 使用 switch 语句,表达式值为 String 类型模拟用户登录操作 */
        final String USER_ACCOUNT = "JohnGao";
        final String USER_PWD = "123456";
        switch (USER_ACCOUNT) {
        case "JohnGao":
            switch (USER_PWD) {
            case "123456":
                System.out.println("用户账号：" + USER_ACCOUNT + "成功登录");
                break;
            default:
                System.out.println("用户账号：" + USER_ACCOUNT + "密码输入错误");
            }
            break;
        default:
            System.out.println("用户账号：" + USER_ACCOUNT + "不存在");
        }
    }
}
```

程序输出如下：

```
用户账号：JohnGao 成功登录
```

上述代码示例中，笔者使用嵌套 switch 语句模拟了一个简单的用户登录操作。虽然与企业级项目中的登录逻辑相比，上述代码显得非常稚嫩（实际的效验逻辑需要考虑的因素较多，比如：登录时安全检查、单点登录机制保障、缓存读写控制以及登录日志信息记录等），但是仅作为一种演示，为大家提供一种效验思路也未尝不可。

A.6　mutil-catch特性

在计算机术语中，异常通常是指程序在运行期间所发生的错误。虽然目前大部分的编程

语言都提供异常处理，但 Java 却为开发人员提供了一种更为简单和结构化的异常处理机制。为什么需要使用异常处理？因为程序的执行结构基于串行化的，也就是说，程序是按照顺序结构从上至下逐一执行的。一旦程序在执行过程中发生异常，后续代码有可能得不到执行机会，从而被迫终止运行。所以我们必须要让程序知道，当发生异常情况后应该如何进行有效的处理。

在 Java 中，所有的异常都通过类来表示。Throwable 是所有异常类型的超类，该类拥有 2 个直接派生类：Exception 和 Error，如图 A-1 所示。

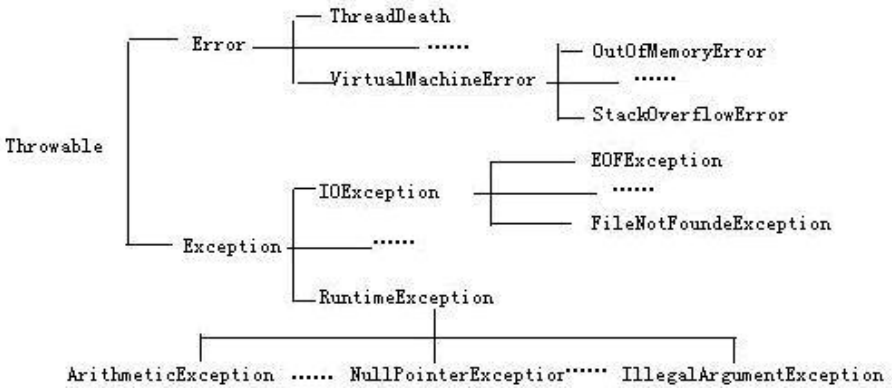


图 A-1 Throwable 的异常派生类结构

Exception 是所有的可恢复性异常类的超类，并且该类型的异常可以划分为运行时异常和一般性异常，那么在程序中如果需要显式地进行手动捕获并处理的异常可以称为非运行时异常，反之则称之为运行时异常。而 Error 是不可恢复性异常类的超类，该类型的异常通常只会发生在 Java 虚拟机内部，比如内存溢出。在此大家需要注意，由于 Error 类型的异常恢复极其困难，因此该类型的异常并不需要在程序中显式地进行捕获与处理。

try、catch、throw、throws 和 finally 等 5 个异常处理关键字构成了 Java 异常处理的子系统，它们彼此之间形成了彼此独立但相关的依赖关系。在 Java 中，异常处理关键字并不能单独使用，也就是说，使用其中任何一个异常关键字，也就同时意味着必须使用另一个异常关键字构成异常处理结构。

在实际的开发过程中，开发人员可以使用 try-cache 语句进行异常的捕获和处理。其中 try 语句内部包含的是需要进行异常监测的代码块，而 catch 子句内部则是包含当异常情况发生时，需要执行的异常处理代码块。如果需要手动向外抛出一个异常，可以使用 throw 关键字。在某些特殊情况下，从一个方法向外抛出一个或者多个异常，则可以使用 throws 关键字。至于 finally 子句则被定义为无论发生任何情况都“必须执行”的特殊代码块。如下所示：

代码 A-14 异常的捕获与处理

```
/**
 * 异常的捕获与处理
 *
 * @author JohnGao
 */
public class ExceptionTest1 {
    public static void main(String[] args) {
        try {
            /* 捕获并处理一个 0 作为除数的异常 */
            System.out.println(100 / 0);
        } catch (ArithmeticException e) {
            System.out.println("0 不能够作为除数进行运算\n" + e);
        }
    }
}
```

程序输出如下：

```
0 不能够作为除数进行运算
java.lang.ArithmeticException: / by zero
```

上述代码示例中，笔者在 catch 子句的表达式内部定义的异常类型为 ArithmeticException，那么当 try 语句内部需要进行异常监测的代码块发生异常时，异常处理则会调用 catch 子句对触发异常的类型进行匹配，如果匹配成功则执行异常捕获和处理操作。

如果 try 语句中有多个异常需要捕获并处理，开发人员要么选择编写多个 catch 子句，要么直接在一个 catch 子句中定义一个 Exception 类型的异常。尽管在实际的开发过程中，选择编写多个 catch 子句进行异常捕获更适用于业务场景，但繁琐 catch 子句编写必然不利于代码后期的阅读和维护，可是直接定义一个 Exception 类型的异常在一个 catch 子句中进行异常捕获，又会因为异常捕获的范围太大而显得过于笼统，并且也不利于异常捕获后的有效处理。那么是否存在一种方式，既能满足异常捕获后的有效处理，同时也有利于代码后期的阅读和维护？随着 Java7 的正式来临，mutil-catch 特性就是为了解决这类问题而诞生的。

mutil-catch 特性允许开发人员减少编写繁琐的 catch 子句和避免定义大范围的异常类型去捕获异常。在程序中一个 catch 子句可以使用符号“|”定义多个异常类型，当程序在运行时发生异常后，异常处理会自动匹配定义在 catch 子句中的多个异常类型。如下所示：

代码 A-15 mutil-catch 语句形式

```
try {
    /* 需要进行异常监测的代码块 */
} catch (ExceptionType1 | ExceptionType2 e) {
```

```
        /* 需要进行异常处理的代码块 */
    } catch (ExceptionType3 | ExceptionType4 e) {
        /* 需要进行异常处理的代码块 */
    }
```

在使用 mutil-catch 特性后，定义在 catch 子句中的异常类型默认都被标记为 final 类型，也就是说不能对其引用进行修改。建议大家不要把所有的异常类型都定义在一个 catch 子句中，因为这样和直接定义 Exception 异常类型无异。在实际开发过程中，我们可以根据业务需要，将相关的一组异常类型定义在一个 catch 子句中，形成一组异常捕获链。而其他异常类型则可以定义到另外一个 catch 子句内，只有充分利用 mutil-catch 的特性，才能够提升编码效率和改善代码结构。如下所示：

代码 A-16 使用 mutil-catch 特性捕获与处理异常

```
    /**
     * 使用 mutil-catch 特性捕获与处理异常
     *
     * @author JohnGao
     */
public class ExceptionTest5 {
    public static void main(String[] args) {
        try {
            /* 调用模拟 I/O 操作或者数据库操作 */
            run();
        } catch (IOException | SQLException e) {
            e.printStackTrace();
        }
    }

    /**
     * 模拟 I/O 操作或者数据库操作
     *
     * @author JohnGao
     *
     * @throws IOException
     *
     * @throws SQLException
     *
     * @return void
     */
    public static void run() throws IOException, SQLException {
        if (Math.random() < 0.1D)
            throw new IOException("程序在执行过程中出现 I/O 操作失败异常");
        else
            throw new SQLException("程序在执行过程中出现数据库访问失败异常");
    }
}
```

程序输出如下：

```
java.sql.SQLException: 程序在执行过程中出现数据库访问失败异常
  at ExceptionTest5.run(ExceptionTest5.java:33)
  at ExceptionTest5.main(ExceptionTest5.java:12)
```

A.7 NIO2.0 文件系统的改变

从大家接触 Java 的第一个 Hello World 程序开始，就已经用到 Java 强大的 I/O 技术，比如 System.out 和 System.in 语句。那么 I/O 技术究竟是指什么呢？其实所谓 I/O 指的就是数据输入和数据输出的过程，我们可以将其称之为 Stream（流，数据通信通道）这个概念。

I/O 技术算得上是开发过程中用得最多的技术之一（不局限于 Java 语言，在任何编程语言中 I/O 技术都被广泛运用），因为几乎无时无刻开发人员都在直接或者间接地使用着 I/O 技术去完成各式各样的需求和功能。从技术层面上来说，使用程序读/写磁盘中某一个指定的文件数据、与数据库建立会话执行 CRUD 操作，甚至使用 Socket 技术进行网络通信等，这些技术底层都需要 I/O 技术作为支撑。除此之外，当我们在使用电脑的时候，几乎从触碰开机键那一刻，I/O 技术就已经开始运作了，比如从硬盘中加载并读取操作系统文件数据、使用浏览器上网冲浪（看视频、听音乐、看新闻）、玩网络游戏或者单机游戏或者使用聊天软件通信等，这些操作都会用到 I/O 技术。

在 Java 中，当需要读取目标数据源的数据时，则需要开启输入流，反之则开启输出流，如图 A-2 和图 A-3 所示。在此需要提醒大家，数据源涉及较广，因此不仅仅只是本地磁盘文件才算作是数据源，内存和网络等载体都可以称之为数据源。

图 A-1 使用程序读取数据源中的数据

图 A-2 使用程序向数据源中写入数据

Java API 中提供了两种类型的数据流，分别是字节流（以 byte 为单位读/写数据）和字符流（以 char 为单位读/写数据）。字节流为处理字节数据的输入/输出提供了方便的方法，常用于读/

写二进制数据。而字符流则为处理字符数据的输入/输出提供了方便的方法，由于字符流采用的是 Unicode 编码格式，因此字符流可以完美地支持国际化。并且在某些情况下，使用字符流相对于字节流而言更为高效，因为字节流是以 byte 为单位，而字符流以 char 为单位，字符流不仅支持单字符的读/写操作，还支持字符数据的整行读/写，这一点是字节流无法做到的。但是这两种数据流各有所长，具体应该使用什么类型的数据流，这还需要根据具体的应用场景而定。

Java 的文件系统主要由 java.io 及 java.nio 两个包内的组件构成。早在 Java7 之前，文件的操作一向都比较麻烦。当然笔者这里提出的麻烦，更多的是指早期 Java API 对文件的管理的不便。比如我们需要编写一个简单的 I/O 程序时，开发人员必然需要编写较多的代码。如下所示：

代码 A-17 传统 I/O 实现的文件复制粘贴操作

```
/* 复制目标数据源数据 */
BufferedInputStream reader = new BufferedInputStream(
      new FileInputStream(COPYFILEPATH));
byte[] content = new byte[reader.available()];
reader.read(content);

/* 将复制数据粘贴至新目录 */
BufferedOutputStream write = new BufferedOutputStream(
      new FileOutputStream(PASTEFILEPATH));
write.write(content);
```

上述代码示例中，仅仅只是编写一个简单的文件复制粘贴逻辑，都需要使用较多的代码。如果你也认同上述程序的繁琐，那么你完全有必要体验下 Java7 对文件系统的一次全新改变。

Java7 推出了全新的 NIO.2 API 以此改变针对文件管理的不便，使得在 java.nio.file 包下使用 Path、Paths、Files、WatchService、FileSystem 等这几个常用类型可以很好地减少开发人员对文件管理的编码量。先从 Path 接口开始讲起，Path 接口的某些功能其实可以和 java.io 包下的 File 类型等价，当然这些功能仅限于只读操作。在实际开发过程中，开发人员可以联合使用 Path 接口和 Paths 类型，从而获取文件的一系列上下文信息。Path 接口常用方法如表 A-1 所示：

表 A-1 Path 接口的常用方法

方法名称	返回值	描述
getNameCount()	int	获取当前文件节点数
getFileName()	java.nio.file.Path	获取当前文件名称
getRoot()	java.nio.file.Path	获取当前文件根目录
getParent()	java.nio.file.Path	获取当前文件上级关联目录

接下来笔者将会开始演示如何使用 Path 接口和 Paths 类型去访问目标文件的上下文信息。如下所示：

代码 A-18　使用 Path 接口与 Paths 类型

```
/**
 * 使用 Path 接口与 Paths 类型获取文件信息
 *
 * @author JohnGao
 */
public class FileTest {
    public static void main(String[] args) {
        Path path = Paths.get("路径: /文件");
        System.out.println("文件节点数:" + path.getNameCount());
        System.out.println("文件名称:" + path.getFileName());
        System.out.println("文件根目录:" + path.getRoot());
        System.out.println("文件上级关联目录:" + path.getParent());
    }
}
```

上述代码示例中，笔者使用 Path 接口和 Paths 类型可以很方便地访问到目标文件的上下文信息。当然这些操作全都是只读的，如果开发人员想对文件进行其他非只读操作，比如文件的创建、修改、删除等操作，则可以使用 Files 类型进行操作。Files 类型常用方法如表 A-2 所示。

表 A-2　Files 类型的常用方法

方法名称	返回值	描述
createFile()	java.nio.file.Path	在指定的目标目录创建新文件
delete()	void	删除指定目标路径的文件或文件夹
copy()	java.nio.file.Path	将指定目标路径的文件复制到另一个文件中
Move()	java.nio.file.Path	将指定目标路径的文件转移到其他路径下，并删除源文件

既然使用传统的 I/O 方式实现一个文件的复制与粘贴简单的操作都显得非常麻烦，那么使用接下来笔者将演示如何使用 Files 类型去实现一个文件的复制与粘贴操作。如下所示：

代码 A-19　Files 类型的使用

```
/**
 * 使用 Files 类型实现文件的复制与粘贴操作
 *
 * @author JohnGao
 */
public class FileTest {
    public static void main(String[] args) {
```

```
            Files.copy(Paths.get("路径:/源文件"), Paths.get("路径:/新文件"));
        }
    }
```

上述代码示例中，使用 Files 类型来管理文件，相对于传统的 I/O 方式来说更加方便和简单，因为具体的操作实现将全部移交给 NIO.2 API，开发人员只需关注与业务逻辑即可。

Java7 还为开发人员提供了一套全新的文件系统功能，那就是文件监测。在此或许有很多朋友并不清楚文件监测有何意义及目的，那么请大家回想下调试成热发布功能后的 Web 容器。当项目迭代后并重新部署时，开发人员无需对其进行手动重启，因为 Web 容器一旦监测到文件发生改变后，便会自动去适应这些“变化”并进行相应的更新操作。Web 容器的热发布功能同样也是基于文件监测功能，所以不得不承认，文件监测功能的出现对于 Java 文件系统来说是具有重大意义的。当然就事论事而言，Java7 的文件监测功能多少存在一些性能和功能上的缺陷，但随着 Java 后续版本的迭代，笔者相信总会有那么一天，能够让那些以口水征服世界的“高手”们停止争论。

如果在程序中需要使用 Java7 的文件监测功能，那么我们务必需要了解 java.nio.file 包下的 WatchService 接口，该接口不仅作为监测服务，还管理着具体的监控细节。开发人员可以通过调用 FileSystems 类型的 newWatchService()方法，从而获取到 WatchService 接口的派生实现。

代码 A-20 获取 WatchService 接口的派生实现

```
/* 获取 WatchService 接口的派生实现 */
WatchService watchService = FileSystems.getDefault()
          .newWatchService();
```

文件监测是基于事件驱动的，事件触发是作为监测的先决条件。开发人员可以使用 java.nio.file 包下的 StandardWatchEventKinds 类型提供的 3 种字面常量来定义监测事件类型，在此大家需要注意，监测事件需要和 WatchService 实例一起进行注册。StandardWatchEventKinds 类型提供的 3 种监测事件，如下所示：

- ☐ ENTRY_CREATE：文件或文件夹新建事件；
- ☐ ENTRY_DELETE：文件或文件夹删除事件；
- ☐ ENTRY_MODIFY：文件或文件夹修改事件。

那么接下来笔者将会演示如何使用 WatchService 类型来实现文件监控。如下所示：

代码 A-21　文件监控

```
/**
 * 使用 WatchService 类型实现文件监控
 *
 * @author JohnGao
 */
public class FileTest {
    public static void main(String[] args) {
        /* 监控目标路径 */
        Path path = Paths.get("C:/");
        try {
            /* 创建文件监控对象 */
            WatchService watchService = FileSystems.getDefault()
                    .newWatchService();
            /* 注册文件监控的所有事件类型 */
            path.register(watchService, ENTRY_CREATE, ENTRY_DELETE,
                    ENTRY_MODIFY);
            /* 循环监测文件 */
            while (true) {
                WatchKey watchKey = watchService.take();
                /* 迭代触发事件的所有文件 */
                for (WatchEvent<?> event : watchKey.pollEvents())
                    System.out.println(event.context().toString()
                            + " 事件类型: " + event.kind());
                if (!watchKey.reset())
                    return;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

上述代码示例中，笔者演示了如何使用 WatchService 接口来实现文件监控，由此可见，使用 WatchService 非常简单和方便。首先我们需要定义好目标监控路径，然后调用 FileSystems 类型的 newWatchService()方法创建 WatchService 对象，接下来还需调用 Path 接口的 register()方法注册 WatchService 实例及监控事件。当这些基础作业层全部准备好后，再编写实时监测循环迭代 WatchKey 来获取所有触发监控事件的文件即可。

在实际的生产环境中，I/O 操作往往会成为性能瓶颈，这是不可否认的事实，也是众多开发人员首要考虑的优化问题。如果在 Windows 环境下我们使用阻塞 I/O 模型来编写分布式应用，其维护的成本往往超出你的想象。因为客户端的链接数量直接决定了服务器内存开辟的线程数量乘以 2（包含一个输入线程和一个输出线程），并且这些线程是无法采取线程池优化的，因为线程的执行时间大于其创建和销毁时间。长时间的大量并发线程挂起，不仅

CPU 要做实时任务切换，其整体物理资源都将一步步被蚕食，直至最后程序崩溃。在早期的网络编程中，采取阻塞 I/O 模型来编写分布式应用，唯一能做的性能优化只有采取传统的硬件式堆机。在付出高昂的硬件成本开销时，其项目的维护性也令开发人员头痛。而且在实际的开发过程中，大部分开发人员会选择将项目部署在 Linux 下运行。跟 Windows 内核结构不同的是，Linux 环境下是没有真正意义上的线程概念。其所谓的线程都采用进程模拟的方式，也就是伪线程。笔者希望大家能够明白，对于并发要求极高的应用场景下，一旦采用阻塞 I/O 模型无疑是非常不可取的。

Java 的 I/O 模型由同步 I/O 和异步 I/O 构成。同步 I/O 模型包含阻塞 I/O 和非阻塞 I/O，而在 Windows 环境下只要调用了 IOCP 的 I/O 模型，就是真正意义上的异步 I/O。IOCP（Input/Output Completion Port，输入/输出完成端口）简单来说是一种系统级的高性能异步 I/O 模型。应用程序中所有的 I/O 操作将全部委托给操作系统线程去执行，直至最后通知并返回结果。Java7 对 IOCP 进行了深度封装，这使得开发人员可以使用 IOCP API 编写高效的分布式应用。在此大家需要注意，由于 IOCP 仅限于使用在 Windows 平台上，因而自然无法在 Linux 平台上使用它（Linux 平台上可以通过 Epoll 模拟 IOCP 实现）。

有过网络编程经验的开发人员都应该明白，在 Windows 平台下性能最好的 I/O 模型是 IOCP，Linux 平台下则是 EPOLL。但是 EPOLL 并不算真正意义上的异步 I/O，EPOLL 只是在尽可能地模拟 IOCP 而已。因为按照 Unix 网络编程的划分，多路复用 I/O 仍然属于同步 I/O 模型，也就是说 EPOLL 其实是属于多路复用 I/O。简单来说异步 I/O 的特征必须满足以下两点：

- □ I/O 请求与 I/O 操作不会阻塞；
- □ 并非程序自身完成 I/O 操作，由操作系统线程处理实际的 I/O 操作，直至最后通知并返回结果。

早在 Java4 版本的时候，NIO（Java New Input/Output，Java 新输入/输出）的出现，使得开发人员可以从阻塞 I/O 的噩梦中彻底挣脱出来。但编写 NIO 的成本较大，学习难度也比较高，使得诸多开发人员望而却步（目前比较成熟的第三方 NIO 产品有 Mina、Netty）。但理解非阻塞 I/O 的原理还是非常有必要的，先来观察下述采用阻塞 I/O 模式编写的分布式应用示例。如下所示：

代码 A-22 使用阻塞 I/O

```
/**
 * 传统网络阻塞 I/O
 *
 * @author JohnGao
 */
```

```
public class IOTest {
    public static void main(String[] args) {
        try {
            ServerSocket server = new ServerSocket(8888);
            Socket clist = server.accept();
            BufferedReader reader = new BufferedReader
                    (new InputStreamReader(clist.getInputStream()));
            /* 未收到 I/O 请求时阻塞 */
            System.out.println(reader.readLine());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

我们可以根据 I/O 的工作内容将其划分为 I/O 请求和 I/O 操作两个部分。上述代码示例中，笔者采用的是阻塞 I/O 模型，可以很明确地发现当客户端成功握手服务端后，如果服务端并没有收到客户端的 I/O 请求，服务端会在 reader.readLine()方法处阻塞。直到成功接收到 I/O 请求后，服务端才会开始执行实际的 I/O 操作。运用阻塞 I/O 模式进行分布式编程，为了保证服务端与客户端集合的成功会话，我们不得不为每一条客户端连接都开辟独立的线程执行 I/O 操作，当然在实际的开发过程中，或许已经没有开发人员会这么做了。

非阻塞 I/O 和阻塞 I/O 最大的不同在于，非阻塞 I/O 并不会在 I/O 请求时产生阻塞，也就是说如果服务端没有收到 I/O 请求时，非阻塞 I/O 会“持续轮循”I/O 请求，当有请求进来后就开始执行 I/O 操作并阻塞请求进程。Java7 允许开发人员使用 IOCP API 进行异步 I/O 编程，这使得开发人员不必再关心 I/O 是否阻塞，因为程序中所有的 I/O 操作将全部委托给操作系统线程去执行，直至最后通知并返回结果。

附录 B

指令助记符[①]

指令	助记符	指令描述
0x00	nop	无操作
0x01	aconst_null	将 null 推送至栈顶
0x02	iconst_m1	将 int 型-1 推送至栈顶
0x03	iconst_0	将 int 型 0 推送至栈顶
0x04	iconst_1	将 int 型 1 推送至栈顶
0x05	iconst_2	将 int 型 2 推送至栈顶
0x06	iconst_3	将 int 型 3 推送至栈顶
0x07	iconst_4	将 int 型 4 推送至栈顶
0x08	iconst_5	将 int 型 5 推送至栈顶
0x09	lconst_0	将 long 型 0 推送至栈顶
0x0a	lconst_1	将 long 型 1 推送至栈顶
0x0b	fconst_0	将 float 型 0 推送至栈顶
0x0c	fconst_1	将 float 型 1 推送至栈顶
0x0d	fconst_2	将 float 型 2 推送至栈顶
0x0e	dconst_0	将 double 型 0 推送至栈顶
0x0f	dconst_1	将 double 型 1 推送至栈顶
0x10	bipush	将单字节的常量值（-128~127）推送至栈顶
0x11	sipush	将一个短整型常量值（-32768~32767）推送至栈顶
0x12	ldc	将 int，float 或 String 型常量值从常量池中推送至栈顶
0x13	ldc_w	将 int，float 或 String 型常量值从常量池中推送至栈顶（宽索引）

① 指令助记符来源于《Java 虚拟机规范（Java SE7 版）》。

续表

指令	助记符	指令描述
0x14	ldc2_w	将 long 或 double 型常量值从常量池中推送至栈顶（宽索引）
0x15	iload	将指定的 int 型本地变量推送至栈顶
0x16	lload	将指定的 long 型本地变量推送至栈顶
0x17	fload	将指定的 float 型本地变量推送至栈顶
0x18	dload	将指定的 double 型本地变量推送至栈顶
0x19	aload	将指定的引用类型本地变量推送至栈顶
0x1a	iload_0	将第一个 int 型本地变量推送至栈顶
0x1b	iload_1	将第二个 int 型本地变量推送至栈顶
0x1c	iload_2	将第三个 int 型本地变量推送至栈顶
0x1d	iload_3	将第四个 int 型本地变量推送至栈顶
0x1e	lload_0	将第一个 long 型本地变量推送至栈顶
0x1f	lload_1	将第二个 long 型本地变量推送至栈顶
0x20	lload_2	将第三个 long 型本地变量推送至栈顶
0x21	lload_3	将第四个 long 型本地变量推送至栈顶
0x22	fload_0	将第一个 float 型本地变量推送至栈顶
0x23	fload_1	将第二个 float 型本地变量推送至栈顶
0x24	fload_2	将第三个 float 型本地变量推送至栈顶
0x25	fload_3	将第四个 float 型本地变量推送至栈顶
0x26	dload_0	将第一个 double 型本地变量推送至栈顶
0x27	dload_1	将第二个 double 型本地变量推送至栈顶
0x28	dload_2	将第三个 double 型本地变量推送至栈顶
0x29	dload_3	将第四个 double 型本地变量推送至栈顶
0x2a	aload_0	将第一个引用类型本地变量推送至栈顶
0x2b	aload_1	将第二个引用类型本地变量推送至栈顶
0x2c	aload_2	将第三个引用类型本地变量推送至栈顶
0x2d	aload_3	将第四个引用类型本地变量推送至栈顶
0x2e	iaload	将 int 型数组指定索引的值推送至栈顶
0x2f	laload	将 long 型数组指定索引的值推送至栈顶
0x30	faload	将 float 型数组指定索引的值推送至栈顶
0x31	daload	将 double 型数组指定索引的值推送至栈顶
0x32	aaload	将引用型数组指定索引的值推送至栈顶
0x33	baload	将 boolean 或 byte 型数组指定索引的值推送至栈顶

续表

指令	助记符	指令描述
0x34	caload	将 char 型数组指定索引的值推送至栈顶
0x35	saload	将 short 型数组指定索引的值推送至栈顶
0x36	istore	将栈顶 int 型数值存入指定本地变量
0x37	lstore	将栈顶 long 型数值存入指定本地变量
0x38	fstore	将栈顶 float 型数值存入指定本地变量
0x39	dstore	将栈顶 double 型数值存入指定本地变量
0x3a	astore	将栈顶引用型数值存入指定本地变量
0x3b	istore_0	将栈顶 int 型数值存入第一个本地变量
0x3c	istore_1	将栈顶 int 型数值存入第二个本地变量
0x3d	istore_2	将栈顶 int 型数值存入第三个本地变量
0x3e	istore_3	将栈顶 int 型数值存入第四个本地变量
0x3f	lstore_0	将栈顶 long 型数值存入第一个本地变量
0x40	lstore_1	将栈顶 long 型数值存入第二个本地变量
0x41	lstore_2	将栈顶 long 型数值存入第三个本地变量
0x42	lstore_3	将栈顶 long 型数值存入第四个本地变量
0x43	fstore_0	将栈顶 float 型数值存入第一个本地变量
0x44	fstore_1	将栈顶 float 型数值存入第二个本地变量
0x45	fstore_2	将栈顶 float 型数值存入第三个本地变量
0x46	fstore_3	将栈顶 float 型数值存入第四个本地变量
0x47	dstore_0	将栈顶 double 型数值存入第一个本地变量
0x48	dstore_1	将栈顶 double 型数值存入第二个本地变量
0x49	dstore_2	将栈顶 double 型数值存入第三个本地变量
0x4a	dstore_3	将栈顶 double 型数值存入第四个本地变量
0x4b	astore_0	将栈顶引用型数值存入第一个本地变量
0x4c	astore_1	将栈顶引用型数值存入第二个本地变量
0x4d	astore_2	将栈顶引用型数值存入第三个本地变量
0x4e	astore_3	将栈顶引用型数值存入第四个本地变量
0x4f	iastore	将栈顶 int 型数值存入指定数组的指定索引位置
0x50	lastore	将栈顶 long 型数值存入指定数组的指定索引位置
0x51	fastore	将栈顶 float 型数值存入指定数组的指定索引位置
0x52	dastore	将栈顶 double 型数值存入指定数组的指定索引位置
0x53	aastore	将栈顶引用型数值存入指定数组的指定索引位置

续表

指令	助记符	指令描述
0x54	bastore	将栈顶 boolean 或 byte 型数值存入指定数组的指定索引位置
0x55	castore	将栈顶 char 型数值存入指定数组的指定索引位置
0x56	sastore	将栈顶 short 型数值存入指定数组的指定索引位置
0x57	pop	将栈顶数值弹出（数值不能是 long 或 double 类型的）
0x58	pop2	将栈顶的一个（long 或 double 类型的）或两个数值弹出（其他）
0x59	dup	复制栈顶数值并将复制值压入栈顶
0x5a	dup_x1	复制栈顶数值并将两个复制值压入栈顶
0x5b	dup_x2	复制栈顶数值并将三个（或两个）复制值压入栈顶
0x5c	dup2	复制栈顶一个（long 或 double 类型的）或两个（其他）数值并将复制值压入栈顶
0x5d	dup2_x1	dup_x1 指令的双倍版本
0x5e	dup2_x2	dup_x2 指令的双倍版本
0x5f	swap	将栈最顶端的两个数值互换（数值不能是 long 或 double 类型的）
0x60	iadd	将栈顶两 int 型数值相加并将结果压入栈顶
0x61	ladd	将栈顶两 long 型数值相加并将结果压入栈顶
0x62	fadd	将栈顶两 float 型数值相加并将结果压入栈顶
0x63	dadd	将栈顶两 double 型数值相加并将结果压入栈顶
0x64	isub	将栈顶两 int 型数值相减并将结果压入栈顶
0x65	lsub	将栈顶两 long 型数值相减并将结果压入栈顶
0x66	fsub	将栈顶两 float 型数值相减并将结果压入栈顶
0x67	dsub	将栈顶两 double 型数值相减并将结果压入栈顶
0x68	imul	将栈顶两 int 型数值相乘并将结果压入栈顶
0x69	lmul	将栈顶两 long 型数值相乘并将结果压入栈顶
0x6a	fmul	将栈顶两 float 型数值相乘并将结果压入栈顶
0x6b	dmul	将栈顶两 double 型数值相乘并将结果压入栈顶
0x6c	idiv	将栈顶两 int 型数值相除并将结果压入栈顶
0x6d	ldiv	将栈顶两 long 型数值相除并将结果压入栈顶
0x6e	fdiv	将栈顶两 float 型数值相除并将结果压入栈顶
0x6f	ddiv	将栈顶两 double 型数值相除并将结果压入栈顶
0x70	irem	将栈顶两 int 型数值作取模运算并将结果压入栈顶
0x71	lrem	将栈顶两 long 型数值作取模运算并将结果压入栈顶
0x72	frem	将栈顶两 float 型数值作取模运算并将结果压入栈顶
0x73	drem	将栈顶两 double 型数值作取模运算并将结果压入栈顶

续表

指令	助记符	指令描述
0x74	ineg	将栈顶 int 型数值取负并将结果压入栈顶
0x75	lneg	将栈顶 long 型数值取负并将结果压入栈顶
0x76	fneg	将栈顶 float 型数值取负并将结果压入栈顶
0x77	dneg	将栈顶 double 型数值取负并将结果压入栈顶
0x78	ishl	将 int 型数值左移位指定位数并将结果压入栈顶
0x79	lshl	将 long 型数值左移位指定位数并将结果压入栈顶
0x7a	ishr	将 int 型数值右（符号）移位指定位数并将结果压入栈顶
0x7b	lshr	将 long 型数值右（符号）移位指定位数并将结果压入栈顶
0x7c	iushr	将 int 型数值右（无符号）移位指定位数并将结果压入栈顶
0x7d	lushr	将 long 型数值右（无符号）移位指定位数并将结果压入栈顶
0x7e	iand	将栈顶两 int 型数值作“按位与”并将结果压入栈顶
0x7f	land	将栈顶两 long 型数值作“按位与”并将结果压入栈顶
0x80	ior	将栈顶两 int 型数值作“按位或”并将结果压入栈顶
0x81	lor	将栈顶两 long 型数值作“按位或”并将结果压入栈顶
0x82	ixor	将栈顶两 int 型数值作“按位异或”并将结果压入栈顶
0x83	lxor	将栈顶两 long 型数值作“按位异或”并将结果压入栈顶
0x84	iinc	将指定 int 型变量增加指定值（i++，i--，i+=2）
0x85	i2l	将栈顶 int 型数值强制转换成 long 型数值并将结果压入栈顶
0x86	i2f	将栈顶 int 型数值强制转换成 float 型数值并将结果压入栈顶
0x87	i2d	将栈顶 int 型数值强制转换成 double 型数值并将结果压入栈顶
0x88	l2i	将栈顶 long 型数值强制转换成 int 型数值并将结果压入栈顶
0x89	l2f	将栈顶 long 型数值强制转换成 float 型数值并将结果压入栈顶
0x8a	l2d	将栈顶 long 型数值强制转换成 double 型数值并将结果压入栈顶
0x8b	f2i	将栈顶 float 型数值强制转换成 int 型数值并将结果压入栈顶
0x8c	f2l	将栈顶 float 型数值强制转换成 long 型数值并将结果压入栈顶
0x8d	f2d	将栈顶 float 型数值强制转换成 double 型数值并将结果压入栈顶
0x8e	d2i	将栈顶 double 型数值强制转换成 int 型数值并将结果压入栈顶
0x8f	d2l	将栈顶 double 型数值强制转换成 long 型数值并将结果压入栈顶
0x90	d2f	将栈顶 double 型数值强制转换成 float 型数值并将结果压入栈顶
0x91	i2b	将栈顶 int 型数值强制转换成 byte 型数值并将结果压入栈顶
0x92	i2c	将栈顶 int 型数值强制转换成 char 型数值并将结果压入栈顶
0x93	i2s	将栈顶 int 型数值强制转换成 short 型数值并将结果压入栈顶

续表

指令	助记符	指令描述
0x94	lcmp	比较栈顶两 long 型数值大小，并将结果（1，0，-1）压入栈顶
0x95	fcmpl	比较栈顶两 float 型数值大小，并将结果（1，0，-1）压入栈顶；当其中一个数值为 NaN 时，将-1 压入栈顶
0x96	fcmpg	比较栈顶两 float 型数值大小，并将结果（1，0，-1）压入栈顶；当其中一个数值为 NaN 时，将 1 压入栈顶
0x97	dcmpl	比较栈顶两 double 型数值大小，并将结果（1，0，-1）压入栈顶；当其中一个数值为 NaN 时，将-1 压入栈顶
0x98	dcmpg	比较栈顶两 double 型数值大小，并将结果（1，0，-1）压入栈顶；当其中一个数值为 NaN 时，将 1 压入栈顶
0x99	ifeq	当栈顶 int 型数值等于 0 时跳转
0x9a	ifne	当栈顶 int 型数值不等于 0 时跳转
0x9b	iflt	当栈顶 int 型数值小于 0 时跳转
0x9c	ifge	当栈顶 int 型数值大于等于 0 时跳转
0x9d	ifgt	当栈顶 int 型数值大于 0 时跳转
0x9e	ifle	当栈顶 int 型数值小于等于 0 时跳转
0x9f	if_icmpeq	比较栈顶两 int 型数值大小，当结果等于 0 时跳转
0xa0	if_icmpne	比较栈顶两 int 型数值大小，当结果不等于 0 时跳转
0xa1	if_icmplt	比较栈顶两 int 型数值大小，当结果小于 0 时跳转
0xa2	if_icmpge	比较栈顶两 int 型数值大小，当结果大于等于 0 时跳转
0xa3	if_icmpgt	比较栈顶两 int 型数值大小，当结果大于 0 时跳转
0xa4	if_icmple	比较栈顶两 int 型数值大小，当结果小于等于 0 时跳转
0xa5	if_acmpeq	比较栈顶两引用型数值，当结果相等时跳转
0xa6	if_acmpne	比较栈顶两引用型数值，当结果不相等时跳转
0xa7	goto	无条件跳转
0xa8	jsr	跳转至指定 16 位 offset 位置，并将 jsr 下一条指令地址压入栈顶
0xa9	ret	返回至本地变量指定的 index 的指令位置（一般与 jsr，jsr_w 联合使用）
0xaa	tableswitch	用于 switch 条件跳转，case 值连续（可变长度指令）
0xab	lookupswitch	用于 switch 条件跳转，case 值不连续（可变长度指令）
0xac	ireturn	从当前方法返回 int
0xad	lreturn	从当前方法返回 long
0xae	freturn	从当前方法返回 float
0xaf	dreturn	从当前方法返回 double
0xb0	areturn	从当前方法返回对象引用

续表

指令	助记符	指令描述
0xb1	return	从当前方法返回 void
0xb2	getstatic	获取指定类的静态域，并将其值压入栈顶
0xb3	putstatic	为指定的类的静态域赋值
0xb4	getfield	获取指定类的实例域，并将其值压入栈顶
0xb5	putfield	为指定的类的实例域赋值
0xb6	invokevirtual	调用实例方法
0xb7	invokespecial	调用超类构造方法，实例初始化方法，私有方法
0xb8	invokestatic	调用静态方法
0xb9	invokeinterface	调用接口方法
0xba	Invokedynamic	调用动态方法
0xbb	new	创建一个对象，并将其引用值压入栈顶
0xbc	newarray	创建一个指定原始类型（如 int，float，char…）的数组，并将其引用值压入栈顶
0xbd	anewarray	创建一个引用型（如类，接口，数组）的数组，并将其引用值压入栈顶
0xbe	arraylength	获得数组的长度值并压入栈顶
0xbf	athrow	将栈顶的异常抛出
0xc0	checkcast	检验类型转换，检验未通过将抛出 ClassCastException
0xc1	instanceof	检验对象是否是指定的类的实例，如果是将 1 压入栈顶，否则将 0 压入栈顶
0xc2	monitorenter	获得对象的锁，用于同步方法或同步块
0xc3	monitorexit	释放对象的锁，用于同步方法或同步块
0xc4	wide	扩展本地变量的宽度
0xc5	multianewarray	创建指定类型和指定维度的多维数组（执行该指令时，操作栈中必须包含各维度的长度值），并将其引用值压入栈顶
0xc6	ifnull	为 null 时跳转
0xc7	ifnonnull	不为 null 时跳转
0xc8	goto_w	无条件跳转（宽索引）
0xc9	jsr_w	跳转至指定 32 位 offset 位置，并将 jsr_w 下一条指令地址压入栈顶